



普通高等院校计算机专业（本科）实用教程系列

# C++语言 基础教程

徐孝凯 编著



1 23456789123123

1 23456789123123



清华大学出版社  
<http://www.tup.tsinghua.edu.cn>



## 【内容简介】

本书是普通高等院校计算机专业(本科)实用教程系列丛书之一,主要介绍 C++语言中常量、变量、表达式、语句、函数等语法成分的定义和使用,数组、字符串、指针、结构、联合、类、文件等数据类型的定义和访问,累加、统计、查找、排序、矩阵运算、数据输入、数据输出等一系列典型程序的设计方法, C++程序结构和 Visual C++ 6.0 集成开发环境的使用等内容。本书包含丰富的例题和练习题,并将配有教学辅导和习题解答光盘,以有利于读者自学。本书是以读者第一次系统学习计算机高级程序设计语言为对象编写的,具有概念定义明确、内容连贯有序、结构层次分明、文字叙述流畅、章节安排合理、深入浅出、方便自学等特点,通过学习本书可以使读者很好地掌握 C++语言的基本语法规则,并具备分析和设计一些典型应用程序的能力。本书适合作为普通高等院校各专业开设程序设计语言课程的教材和教学参考书。



# 目 录

第一章 C++ 语言概述 .....	1
1.1 引言 .....	1
1.2 C++ 字符集 .....	2
1.3 C++ 单词 .....	3
1.4 C++ 语句 .....	5
1.5 C++ 函数 .....	7
1.6 C++ 程序 .....	10
1.7 VC++ 6.0 集成开发环境简介 .....	12
习题一 .....	16
第二章 数据类型和表达式 .....	19
2.1 数据类型 .....	19
2.2 常量 .....	21
2.2.1 整型常量 .....	22
2.2.2 字符常量 .....	23
2.2.3 逻辑常量 .....	24
2.2.4 枚举常量 .....	24
2.2.5 实型常量 .....	25
2.2.6 地址常量 .....	26
2.3 变量 .....	26
2.4 运算符 .....	30
2.5 函数 .....	38
习题二 .....	42
第三章 流程控制语句 .....	50
3.1 if 语句 .....	50
3.2 switch 语句 .....	53
3.3 for 语句 .....	58
3.4 while 语句 .....	67
3.5 do 语句 .....	72
3.6 跳转语句 .....	77
习题三 .....	81

<b>第四章 数组和字符串</b>	90
4.1 数组的概念	90
4.2 数组的定义	91
4.2.1 一维数组	91
4.2.2 二维数组	95
4.2.3 使用 typedef 语句定义数组类型	99
4.3 数组的应用	101
4.3.1 数值计算	101
4.3.2 统计	105
4.3.3 排序	106
4.3.4 查找	109
4.4 字符串	112
4.4.1 字符串概念	112
4.4.2 字符串函数	115
4.4.3 字符串应用举例	118
习题四	121
<b>第五章 指针</b>	128
5.1 指针的概念	128
5.2 指针变量	129
5.3 指针运算	134
5.4 指针与数组	138
5.4.1 指针与一维数组	138
5.4.2 指针与二维数组	140
5.5 引用变量	141
5.6 动态存储分配	143
习题五	146
<b>第六章 函数</b>	151
6.1 函数的定义	151
6.1.1 定义格式	151
6.1.2 定义格式举例	151
6.1.3 有关函数定义的几点说明	152
6.2 函数的调用	155
6.2.1 调用格式	155
6.2.2 调用过程	156
6.2.3 函数调用举例	158
6.3 变量的作用域	162
6.3.1 作用域分类	162

6.3.2 程序举例 .....	164
6.4 递归函数 .....	169
6.5 函数重载 .....	173
6.6 函数模板 .....	174
6.7 函数指针 .....	178
习题六 .....	180
<b>第七章 结构与联合 .....</b>	<b>190</b>
7.1 结构的定义 .....	190
7.1.1 结构定义格式 .....	191
7.1.2 定义格式举例 .....	191
7.1.3 结构使用说明 .....	192
7.2 结构变量的定义和初始化 .....	193
7.2.1 用结构类型名定义变量 .....	193
7.2.2 定义结构类型的同时定义变量 .....	195
7.2.3 定义无名结构类型的同时定义变量 .....	195
7.3 结构成员的访问操作 .....	196
7.4 使用结构的程序举例 .....	197
7.5 结构与函数 .....	206
7.6 结构与链表 .....	210
7.7 结构与操作符重载 .....	213
7.8 联合 .....	221
7.8.1 联合的定义和访问 .....	221
7.8.2 使用联合举例 .....	223
习题七 .....	228
<b>第八章 类与对象 .....</b>	<b>232</b>
8.1 类的定义 .....	232
8.1.1 类的定义格式 .....	232
8.1.2 定义格式举例 .....	233
8.1.3 有关说明 .....	235
8.2 构造函数 .....	240
8.2.1 无参构造函数和带参构造函数 .....	240
8.2.2 拷贝构造函数 .....	245
8.2.3 赋值重载函数 .....	247
8.2.4 构造函数中的初始化表 .....	247
8.3 析构函数 .....	248
8.4 友元函数和友元类 .....	250
习题八 .....	256

第九章 类的继承与多态性 .....	266
9.1 类的继承 .....	266
9.1.1 派生类定义的格式 .....	266
9.1.2 格式举例 .....	267
9.1.3 应用举例 .....	274
9.2 类的虚函数与多态性 .....	277
9.3 类的静态成员 .....	281
9.4 类模板 .....	283
习题九 .....	285
第十章 C++ 流 .....	290
10.1 C++ 流的概念 .....	290
10.2 输入输出格式控制 .....	292
10.2.1 ios 类中的枚举常量 .....	292
10.2.2 ios 类中的成员函数 .....	293
10.2.3 格式控制操纵符 .....	296
10.3 文件操作 .....	298
10.3.1 文件的概念 .....	298
10.3.2 字符文件的访问操作 .....	302
10.3.3 字节文件的访问操作 .....	309
10.4 字符串流 .....	315
习题十 .....	317
附录 ASCII 代码表 .....	322



# 第一章 C++ 语言概述

## 1.1 引言

C++ 语言是目前世界上最流行和应用范围最广的一种计算机高级程序设计语言。它从早期的 C 语言逐渐发展演变而来, C++ 对 C 语言不是简单的扩充, 而是从面向过程的语言发展为既面向过程又面向对象的语言, 以适应软件开发技术从面向过程转向面向对象的客观发展的需求。

面向过程是求解问题的一种传统方法, 它把整个问题按功能划分为若干个相对独立的小问题, 每个小问题又可以按功能划分为若干个相对独立的更小问题, 依此类推, 直到最低一层的问题较容易用一种计算机语言编写的程序模块实现为止。在面向过程的程序设计中, 每个程序模块都具有一定的相对独立的功能, 通过较小的程序功能模块的组合就可以形成较大的程序功能模块, 最后形成一个完整的程序。在采用面向过程的方法进行程序设计时, 整个程序的功能是通过程序模块之间的相互调用完成的, 若问题比较复杂, 程序结构即模块之间的调用关系很容易变得复杂和混乱, 并且也容易增加模块之间的依赖性以及调试和修改程序的难度。

面向对象是求解问题的一种新的思路和方法, 它把求解问题中的所有事物(即独立个体)都看做为各自不同的对象, 进而把具有共同特征的对象归属为一个类, 由此得到若干个不同的类, 每个类是对该类事物(对象)的抽象描述, 通过相同或不同类对象之间的相互作用和通信使问题得以解决。

面向过程方法基于对实现某一功能所进行的操作过程的描述, 当功能发生变化时, 哪怕是稍微变化, 都需要重新修改和调试对应的程序模块; 面向对象方法基于对类和对象的描述, 包括对对象属性和进行所有操作的描述, 当对象发生变化时, 可以让原对象保持不变, 再另外建立新对象, 让它继承原对象, 并根据发生的变化定义出新的部分, 此时只需要编写和调试这新增的部分, 对原对象中的代码不需要做任何改变而全部继承下来。这种改良措施更符合人们的思维习惯和解决问题的方法, 同面向过程方法相比, 可以大大减少软件编写和调试的工作量, 提高软件的复用性、可靠性和可扩充性, 缩短软件更新周期。

C++ 语言是对 C 语言的继承、丰富和发展, 既适合开发面向过程的程序, 又适合开发面向对象的程序。无论是利用 C++ 语言开发面向过程的程序, 还是开发面向对象的程序, 都需要掌握 C++ 语言中的一些基本内容, 如数据类型、常量、变量、表达式的含义与使用, 函数的定义与调用, 文件操作, 各种语句的格式与功能, 程序的基本结构等。若在此基础上再深入学习和掌握有关“类”的知识, 就可以利用面向对象的程序设计方法进行软件开发。

由于 C++ 语言系统庞大, 对于初学计算机高级程序设计语言的读者来说, 最好把它分为两个阶段, 作为两门课程学习, 第一阶段主要学习 C++ 语言基础知识, 第二阶段主要学习

与“类”有关的进行面向对象的程序设计知识,这后一阶段最好不要就语言学语言,而要结合数据结构或软件工程课程一起学习效果最佳。

C++ 语言有标准版本,但各软件公司开发的C++ 语言版本并不是严格遵守它,而是与它兼容且稍有修改和扩充。现在普遍使用的是 Microsoft 公司的 Visual C++ 6.0 版本和 Borland 公司的C++ Builder 3.0 及以上版本,它们都是在 Windows 操作系统环境下运行的可视化集成开发工具。本书以 Visual C++ 6.0 为蓝本,向读者介绍C++ 语言的基本内容和进行一般算法设计的知识,使读者能够进行模块化和结构化的面向过程的程序设计,以及较简单的面向对象的程序设计,为学习后续数据结构课程和进行更复杂的算法设计奠定基础。

自然语言(如汉语、英语等)是人们进行交流的工具,它的基本符号是字或字符,由字(字符)构成基本词法单位——单词,由单词构成句子,由句子组成文章。计算机语言同自然语言一样,也具有字、词、句、章的体系结构。如对于C++ 语言来说,它具有规定的字符集,由一个或若干个字符按照词法规则构成C++ 单词,由一个或若干个单词按照语法规则构成C++ 语句,而完成某一功能的一条或若干条语句被定义为一个程序模块——函数,由一个或若干个函数以及一些其他的语法成分构成一个独立的程序文件,由一个或若干个程序文件构成一个完整的C++ 程序。

## 1.2 C++ 字符集

下面分类列出在C++ 语言中规定的全部字符。

### 1. 大、小写英文字母(52 个)

A~Z, a~z

注意:在 C/C++ 语言中,同一字符的大写和小写被视为不同的字符,这与其他语言中的规定不同。如 A1b 和 a1b 或 ABC 是完全不同的标识符。

### 2. 十进制数字符号(10 个)

0~9

### 3. 标点符号(8 个)

, 逗号	// 数据之间的分隔符
; 分号	// 简单语句结束符
' 单引号	// 字符常量起、止标记符
" 双引号	// 字符串常量起、止标记符
: 冒号	// 语句标号结束符或条件运算符
空格	// 语句中各成分之间的分隔符
{ 左花括号	// 复合语句的开始标记
} 右花括号	// 复合语句的结束标记

#### 4. 单字符运算符(19个)

( 左圆括号 // 简称左括号,同右圆括号配对使用,用于表达式和函数运算  
) 右圆括号 // 简称右括号  
[ 左中括号 // 左、右中括号必须配对使用,用于数组元素访问  
] 右中括号  
+ 加号或正号  
- 减号或负号  
\* 乘号或间接访问运算符  
/ 除号  
% 取整余数  
. 小数点或结构成员访问符  
< 小于号或左尖括号  
= 赋值号  
> 大于号或右尖括号  
! 感叹号 // 逻辑非运算符  
~ 波折号 // 按位取反运算符  
& // 取地址或按位与运算符,同时又是引用说明符  
^ 尖字符 // 按位异或运算符  
| 竖线 // 按位或运算符  
? 问号 // 条件表达式运算符

#### 5. 特殊用途符号(3个)

# 井字符 // 预处理命令行的开始标记  
\ 反斜线 // 转义字符序列的开始标记  
\_ 下划线 // 只用于标识符中

**注意:**在字符串常量(简称字符串)中能够使用任何 ASCII 码字符。如 \$ 不属于 C++ 字符,但可以使用在 C++ 字符串中。另外,在汉字操作系统的支持下,任何汉字区位码字符均可以使用在 C++ 字符串中。如“汉字”、“24.56\$”、“a+b=”等都是合法的字符串。

### 1.3 C++ 单词

由 C++ 字符按照一定的组词规则可以构成各种 C++ 单词,可以把 C++ 单词分为以下五类。

#### 1. 保留字

保留字是 C++ 系统预定义的、由小写英文字母组成的单词、词头或词组,VC++ 6.0 中

的全部保留字如下:

auto	bool	break	case	char	class	const
continue	default	delete	do	double	else	enum
extern	false	float	for	friend	goto	if
inline	int	long	new	operator	private	protected
public	register	return	short	signed	sizeof	static
struct	switch	template	this	true	typedef	union
unsigned	virtual	void	volatile			

每个保留字都被系统赋予了一定的含义,具有了相应的功能,所以用户不能再将它们作为非保留字使用。

另外,在预处理命令中,其命令关键字虽然不算作C++保留字,但也最好把它们看做为C++保留字,不要使用它们作为其他用途,以免引起混乱。这些命令关键字有 include、define、ifdef、ifndef 等。

## 2. 标识符

标识符是用户在程序设计中给特定量所起的名字。在C++语言中规定:每个标识符必须是由英文字母、十进制数字符号和下划线组成的一串字符,并且第一个字符必须是英文字母或下划线。每个标识符中的字符数可以任意,但只有前 32 个字符有效。通常,一个标识符由 1~32 个字符所组成。如 a、ab、size、Max、x1、y25、fun\_1 等都是合法的标识符,而 3xy、“work”、lable:、Hi-4、list length 等都是非法的标识符,因为第一个标识符以数字开头,其余四个标识符中均使用了非法字符。

给一个特定量命名一个标识符时,为了便于记忆和阅读,最好使用该特定量的英文或汉语拼音作为标识符,有时将第一个字母大写,有时使用下划线连接两个英文或拼音单词。如可以用 wages、wage、Wage、Gongzi 等表示工资,用 Name、xingning、XM 等表示姓名,用 maxWage、MaxWage、max\_wages 等表示最高工资。

在C++程序设计中,变量、对象、符号常量、用户定义的数据类型、函数等的名字都是需要用户定义的标识符。

## 3. 常量

常量分为数值常量、字符常量和字符串常量三类。日常使用的十进制常数可以直接作为C++数值常量使用。如 32、-128、3.26、+100、-50.718 等都是合法的C++数值常量,简称常数。

字符常量就是单个 ASCII 码字符,表示时为了把它同单个字符的标识符或数值相区别,必须用单引号括起来。如 'a'、'B'、'+'、';'、'5' 等都是字符常量。使用以单引号括起来的转义字符序列也可以表示字符常量,特别是用来表示像回车、换行等控制字符常量,这些将在以后介绍。

字符串常量就是由 ASCII 码字符和汉字区位码字符组成的一串字符,同样,为了把它同其他语法成分相区别,表示时必须用双引号括起来。如 "a+b="、"main:"、"x,y,z="、"Not found"、"length of table"、"1. 输出线性表的长度"等都是字符串常量,简称为字符串。

由以上讨论可知:a、'a'和"a"是完全不同的,它们分别为一个标识符、一个字符常量和一



个字符串。

#### 4. 运算符

运算符是对数据进行运算的符号。C++ 运算符有单字符运算符,如 +、-、\*、/等,有双字符运算符,如 <=、!=、->、++、&&、\*= 等,也有三字符运算符,如 <<=、>>= 等,另外还有三个保留字运算符 new、delete 和 sizeof。以后将详细列出所有 C++ 运算符,并一一介绍其具体功能。

由运算符和操作数可以构成各种表达式,对表达式进行计算的结果通常得到一个确定的值。如  $25 * 4 + 15$  就是一个数值表达式,其求值结果为 115。当然一个单独的操作数也可以看做为一个表达式,因为它本身就是一个值或对应一个值。如一个常数 25,一个变量 x,一个字符 '!',一个字符串 "apple"等操作数都是表达式,不过它们为最简单的表达式。

#### 5. 标点符号

C++ 字符集中列出的每个标点符号可以单独作为 C++ 单词使用,作为一个语法成分出现在语句中。

### 1.4 C++ 语句

由 C++ 单词按照一定的语法规则排列起来就形成语句。虽然每种语句的语法规则不同,但除了复合语句外,最后都必须以分号结束。下面的每一行都是一条 C++ 语句,当然其前面的编号除外。

- (1) int x;
- (2) x = 20 \* 35 - 6;
- (3) if(x >= 100) cout << x;
- (4) break;
- (5) typedef int DataType;
- (6) void Sort(int aa[], int nn);
- (7) Sort(a, n);

在第一条语句中,含有 4 个单词(即语法成分),依次为表示整型的保留字 int、空格、表示变量的标识符 x 和分号。在第二条语句中含有 8 个语法成分,其中为 1 个变量标识符,3 个运算符,3 个常数和 1 个分号。第三至第七条语句中,分别含有 10 个、2 个、6 个、15 个和 7 个单词,请读者验证。

按照语句功能,可以把 C++ 语句分为以下八类:

#### 1. 类型定义语句

类型定义语句又称类型说明语句,用来定义系统预定义类型之外的、用户需要使用的数据类型。如结构、联合、枚举和类类型都需要用户结合应用情况具体定义。

## 2. 变量定义语句

变量定义语句又称变量说明语句,用来定义程序中需要使用的属于某个类型的变量。如上述第一条语句就定义了整型变量  $x$ ,其中 `int` 表示系统预定义的整数类型, $x$  为一个变量标识符,以后可以用它来表示(即保存)一个整数。在上述第二条语句中,用  $x$  保存一个整数 694,它是赋值号右边表达式  $20 * 35 - 6$  的值。

## 3. 函数原型语句

它是用以声明一个函数存在并指定调用格式的语句。通常情况下,函数原型语句出现在一个程序或程序文件的开始部分,以便在其后的函数中能够调用该函数,而该函数原型语句所对应的函数定义可以出现在整个程序中的任何位置,甚至可以不在同一个程序文件中。如上述第六条语句就是一条函数原型语句,调用该函数时要使用两个参数,一个为整型数组,另一个为整型数,该函数执行后不返回任何值。

## 4. 表达式语句

任何一个 C++ 表达式后加上一个语句结束符分号就成为一条语句,称此为表达式语句。最常用的表达式语句为赋值表达式语句和函数调用表达式语句。上述第二条为赋值表达式语句,第七条为函数调用表达式语句。

## 5. 复合语句

用花括号括起来的语句序列合起来称为一条语句,即复合语句。如 `|int x=3,y=4; y=x+2*y;|` 就是一条复合语句,它包含有两条简单语句。一条复合语句中可以包含任意多条语句,包括不含任何语句,并且每条语句可以是任何种类的语句,包括仍可以是复合语句。如 `|}|,|;|,|x=40;|` 等都是合法的复合语句,其中第一条中不含有语句,第二条中含有一条空语句,即只有分号的语句被称为空语句,它也是一条合法的语句,第三条中含有一条简单语句。

**注意:** 复合语句是以左花括号作为开始标记,右花括号作为结束标记的,其后不需要使用分号,若误用分号,编译系统则认为是后接一条空语句。如 `"|int a; a=1;|;"` 为两条语句,前者为复合语句,后者为空语句。

由于复合语句也是一条语句,所以,以后所说的语句均指所有种类的语句,当然包括复合语句在内。

## 6. 选择语句

选择语句具有根据条件选择是否执行某条语句的功能,或者具有根据条件从多个入口点中选择某个入口点执行的功能。如上述第三条语句就是一条选择语句,它根据条件  $x \geq 100$  是否成立来决定是否执行把  $x$  的值输出到屏幕上的操作,若条件成立则输出,否则不输出。

## 7. 循环语句

循环语句具有根据条件控制一条语句重复执行的功能。如“while(i++ <= 10) x = x + i;”就是一条循环语句,它将使内含的“x = x + i;”语句反复执行,直到条件 i + <= 10 不成立为止。

## 8. 跳转语句

计算机执行程序时都是按照语句出现的先后次序,从上向下依次执行的,而跳转语句能够改变这种执行次序,使执行转移到其他位置,接着从这个位置起向下执行。

在以上叙述的八类语句中,前三类属于说明性语句,后五类属于执行性语句。当对程序进行编译生成目标代码文件时,在目标代码文件中只存在执行性语句所对应的目标代码,不存在说明性语句所对应的目标代码。

在上述每类语句中,通常还包含有多种具有不同语句格式和用法的语句,关于它们的详细内容将在以后章节中陆续讨论。

# 1.5 C++ 函数

C++ 函数包括系统函数和用户函数两大类。

## 1. 系统函数

系统函数又叫标准函数、预定义函数、库函数等,顾名思义,它是由C++语言系统本身提供给用户使用的,是一系列具有各自用途的函数的总称。如“int abs(int x);”就是一个系统函数,其功能是返回整型参数 x 的绝对值。C++ 中的所有系统函数被分类组织在C++系统层次目录中的 lib 子目录里的相应库函数文件中,而所有系统函数的原型(即原型语句)被分类组织在C++系统层次目录中的 include 子目录里的相应函数头(即函数原型)文件中,每个头文件均以 .h 作为扩展名。如 math.h 头文件中保存有常用的数学函数原型, string.h 头文件中保存着对字符串操作的常用函数原型, iostream.h 头文件中保存着对标准输入/输出设备(即键盘/显示器屏幕)进行数据操作的函数原型及一些输入/输出流类和对象, fstream.h 头文件中保存着对外部文件(通常为磁盘文件)进行数据操作的函数原型及一些文件流类和对象。

当一个程序中需要使用某个系统函数时,必须在程序文件的开始部分写上预处理包含命令,把该函数原型所属的头文件包含进去。

**注意:**所有预处理命令都是以 # 字符作为开始标记的,并且每条预处理命令单独占有一行。每条预处理包含命令以 # 字符后接 include 关键字开始,其命令格式为:

```
#include <头文件名>
```

该命令的关键字后是用一对尖括号括起来的头文件名,在该头文件中应该包含需调用函数的原型。

**注意：**预处理命令行不是语句，所以不能以分号结束。预处理包含命令的另一种格式为：

```
# include "头文件名"
```

程序中的每条预处理命令是在程序编译过程的前期阶段(又称为预处理阶段)中被处理的，处理结束后该命令将被删除掉，在程序编译过程的后期阶段以及以后的连接和执行过程中将不存在预处理命令。对于每条预处理包含命令来说，其处理过程是把该命令置换为头文件名所指定的头文件中的全部内容。

对于上述给出的两种包含命令格式，系统处理时的查找头文件的路径有所不同。对于第一种格式，将从C++系统层次目录中查找头文件，若查找不到则给出错误信息；对于第二种格式，将首先从操作系统的当前工作目录中查找头文件，若查找不到，再接着从C++系统层次目录中查找头文件，若再查找不到，则给出错误信息。另外，在第二种格式中，头文件名可以是带有磁盘号和路径名的完整的文件标识符，此时将从指定路径中查找头文件，若找不到再从C++系统层次目录中查找。

C++头文件可以由C++系统提供，也可以由用户建立，通常由用户建立的头文件存入用户在用户指定的磁盘目录中。若使用的头文件由系统提供，则采用第一种包含命令格式，若由用户建立，则采用第二种包含命令格式。由用户建立的头文件通常包含有用户函数原型，符号常量定义，数据类型定义等语句。

由于一个# include命令只包含一个头文件，所以一个程序文件需要使用多少个头文件就需要使用多少条包含命令。

除了可以在# include命令中使用头文件外，通常还可以使用一个程序文件。如：

```
# include "abc.cpp"
```

就把abc.cpp程序文件包含到所在的文件中，使该程序文件内容成为所在文件的一部分。

## 2. 用户函数

用户函数是用户根据解决问题的需要而编写出的一个具有相对独立功能的一个程序单元，又称为程序模块或功能模块。在一个完整的程序中可以包含一个或任意多个用户函数，但必须有并且只有一个规定起名为main的用户函数，它是一个具有特殊地位的用户函数，称为程序的主函数，该函数是由计算机操作系统在程序运行时被自动调用执行的，而其他函数(包括用户函数和系统函数)则是通过它直接或间接调用执行的。运行一个C++程序的过程就是让操作系统自动调用执行main主函数的过程。

下面是一个最简单的C++程序：

```
#include<iostream.h>
void main(void)
{
    cout << "This a simple C++ program." << endl;
}
```

该程序第一行为预处理包含命令行，它包含有iostream.h这个系统头文件，在下面主函数中使用的cout和endl标识符在此头文件中均有定义，使用的运算符<<在此头文件中也



有重载定义。iostream.h 头文件也可以用 iomanip.h 来代替,这个头文件包含有 iostream.h 头文件的全部内容,并且还包含有控制数据输入/输出格式的其他内容。

程序中的第 2~5 行为主函数,它的函数类型,又称函数返回值类型为 void,即空,表示不返回值,main 为主函数名,其后为用一对圆括号括起来的参数表,该参数表中的具体内容为 void,表示参数表为空,即不带任何参数。当然,若不在圆括号内写明 void,则也表示参数表为空。参数表后的从左花括号开始到右花括号结束的部分称为该函数的函数体,它是一条复合语句,其中只含有一条语句,它是一条表达式语句,cout 标识符代表标准输出设备显示器,运算符 << 表示把右边的一个数据项的值输出到左边所代表的设备中,在此例中则是把一个字符串输出到显示器屏幕的一个显示窗口上显示出来。运算符 << 可以在表达式中使用多次,每次都是把右边的一个数据项的值输出到最左边的标识符所代表的设备中。该语句最后使用的标识符 endl 是一个符号常量,即用标识符表示的常量,其值为一个换行符。当系统执行这条语句时,首先从屏幕上 C++ 显示窗口的当前光标位置(开始位于窗口的左上角)起显示出双引号内的字符串,接着再输出一个换行符,使当前光标移到下一行开始位置,即最左边位置。该程序运行时将在显示窗口显示出如下内容:

```
This a simple C++ program.  
Press any key to continue
```

其中第二行内容是系统在运行结束前自动给出的,提示用户在按下任一键后退出运行状态,返回到 C++ 系统集成开发环境中。

一个 C++ 函数的定义格式如下:

[<函数类型>] <函数名> ([<参数表>]) <复合语句>

其中每对尖括号连同内部的汉字注释表明为一个语句成分,一对中括号表示其中的语句成分可以省略。<函数类型> 为系统预定义或用户已定义的数据类型,用以表示函数被调用执行后返回值的类型,若省略该项,则隐含为系统预定义的 int 类型。<函数名> 是用户为该函数所起的名字,是由用户命名的一个合法标识符。<参数表> 是用逗号分开的一组变量说明,用来保存从调用该函数的实参表中传送来的值,若省略该项,则参数表为空。<复合语句> 是用一对花括号括起来的语句序列,它可以书写在多行上,每一行可以书写一条或多条语句,每一条语句也可以单独占一行或分为多行书写。程序中每一行中的所有空格符、制表符(对应键盘上的 Tab 键)和最后隐含的回车换行符(对应键盘上的 Enter 键)只作为成分之间的分隔符,所以成分之间使用一个或任意多个这样的字符对语句格式没有影响,只影响其屏幕显示效果。

在一个函数定义中,最后的语句成分 <复合语句> 称为该函数的函数体,除去函数体的剩余部分称为该函数的函数头。一个函数的函数头就是该函数的原型,其原型语句可看做为是用函数头后加分号的表示。

下面看一个用户函数定义的例子:

```
#include <iomanip.h>  
int Add(int x, int y)  
{  
    int z = x + y;  
    return z;  
}
```

```
}
```

该函数的功能是实现两个整数相加并返回其和。函数的返回类型为整型,即 `int`,函数名为标识符 `Add`,参数表中包含有两个参数说明,把 `x` 和 `y` 均说明为整型。函数体中包含有两条语句,第一条语句定义了一个整型变量 `z`,并把表达式 `x + y` 的值赋给它,第二条语句称为返回语句,该语句的关键字(它必然是一个 C++ 保留字)为 `return`,该语句的功能是结束该函数的执行过程并把关键字后表达式的值返回到调用该函数的位置。

假定 `a` 是一个整型变量,调用该函数的表达式语句为:

```
a = Add(3,4);
```

计算机执行该语句时,首先调用执行 `Add` 函数,把 3 和 4 分别传送给参数 `x` 和 `y`,执行函数体时把 `x + y` 的值 7 赋给变量 `z`,把 `z` 的值 7 作为函数值返回;接着把返回值 7 赋给变量 `a`,使 `a` 的值变为 7。

## 1.6 C++ 程序

一个 C++ 程序由一个主函数和若干个用户函数,以及一些预处理命令行,一些数据类型、符号常量、变量的定义所组成。它们可以保存到一个文件中,也可以分开保存到多个文件中,每个文件的文件名当首次保存文件时由用户提供,文件的默认扩展名为 `.cpp`。

下面是一个 C++ 程序的例子,可以把它输入保存到一个文件中,假定该文件标识符为 `d:\job\prac.cpp`,即保存到 d 盘 `job` 目录下,其文件主名用 `prac` 表示,扩展名采用默认扩展名。

```
#include <iostream.h>
int big(int x, int y);
void main()
{
    int a,b,c;
    cout << "输入 a 和 b 的值: ";
    cin >> a >> b;
    c = big(a,b);
    cout << "a,b,c=" << a << ', ' << b << ', ' << c << endl;
    cout << "重新输入 a 和 b 的值: ";
    cin >> a >> b;
    c = big(a,b);
    cout << "a,b,c=" << a << ', ' << b << ', ' << c << endl;
}
int big(int x, int y)
{
    if(x >= y) return x;
    else return y;
}
```

该程序包含有两个函数,一个为主函数,另一个为 `big` 函数。`big` 函数的功能是求出两个参数 `x` 和 `y` 中较大的值并作为函数值返回。该函数的函数体只含有一条语句,即条件语

句,该语句关键字为 if 和 else,该语句中的两条返回语句都属于它的子句,该语句执行时首先判断关系表达式  $x \geq y$  是否成立,若成立则返回 x 的值,否则返回 y 的值。

程序第一行为预处理包含命令行,包含有 iostream.h 头文件,使得在后面的函数中能够使用键盘和显示器进行数据输入输出操作。

程序第二行为 big 函数的原型语句,通过它就可以知道该函数的参数个数、每个参数的类型,函数的返回类型等信息。

一个函数的原型语句,除了最后的分号外与该函数定义中的函数头是相同的,不过有一点例外,那就是在原型语句的函数参数表中,每个参数的变量名可以与函数头中的对应变量名不同,或者可以省去不用。如下面的两条语句都是 big 函数的原型语句,它们与程序中的原型语句等效。

```
int big(int a, int b);  
int big(int, int);
```

程序的第 3~14 行为主函数,其中第 3 行为函数头,其余为函数体。函数体中的第一条语句为变量定义语句,定义了 a、b 和 c 这 3 个整型变量;第二条语句为表达式语句,它向屏幕输出运算符 << 后的字符串,由于该语句是以标识符 cout 开头的,所以通常被称之为标准输出语句,简称输出语句或 cout 语句;第三条语句也为一条表达式语句,它从键盘上输入得到两个整数并分别赋给 a 和 b 变量中,由于该语句是以代表标准输入设备键盘的标识符 cin 开头的,其功能是从键盘上给变量输入数据,所以通常被称之为标准输入语句,简称输入语句或 cin 语句;第四条语句是一条赋值表达式语句,简称赋值语句,它利用 a 和 b 调用 big 函数并把返回值赋给变量 c;第五条语句为输出语句,依次输出每个运算符 << 后的数据项的值;第 6~9 条与上述相应语句的功能相同。

程序的 16~20 行为 big 函数的定义,其中第 16 行为函数头,其余为函数体。

假定运行该程序时,首先从键盘上输入 12 和 25 这两个整数并按下回车键,接着输入 38 和 16 这两个整数并按下回车键,则得到的屏幕显示结果如下:

```
输入 a 和 b 的值: 12 25  
a, b, c = 12, 25, 25  
重新输入 a 和 b 的值: 38 16  
a, b, c = 38, 16, 38  
Press any key to continue
```

在这个程序中使用的 cin 同 cout 一样,都是在 iostream.h 中定义的,用于分别表示标准输入设备键盘和标准输出设备显示器的标识符,使用的 >> 也同 << 一样都是在 iostream.h 中有重载定义的,分别称为提取和插入运算符,它们分别用于从键盘输入(即提取)数据和向显示器输出(即插入)数据。在一条输入语句中,运算符 >> 可以被使用多次,每次使用能够从键盘上为它右边的一个变量输入数据。

**注意:**当程序执行到 cin 语句时,若存放键盘输入数据的缓冲区为空(程序开始运行时总是空的),则暂停向下执行,等待用户从键盘向缓冲区输入数据,每个数据之后必须以空白符(即空格、制表符或回车换行符)结束,并且只有按下回车键(对应回车换行符)后才会把一行数据送入到缓冲区中,当缓冲区中有足够的数据供 cin 语句读取给变量时,就执行给每个

变量依次读取一个数据的操作,读取结束后表明该语句执行完毕,接着继续向下执行。

在一个程序文件中,若函数的定义出现在调用它的位置之前,则可省去该函数的原型语句,因为当调用该函数时,系统能够从函数定义的函数头中获得函数原型信息。对于上述程序文件,若把 big 函数放在 main 函数之前定义,则可省去第 2 行的原型语句,当然继续使用它也是正确的,并且是较好的选择。

这里顺便介绍一下如何在程序中使用注释的问题。在一个源程序文件或头文件中,可以在任何需要的地方加上注释内容以增强程序的可读性。C++ 有两种加注释的方法:一是使用双斜线,从它开始到行尾的所有内容都被看做为注释,此方法只能使用在一个程序行中所有语法成分之后;二是使用双字符/\* 作为注释开始和使用双字符\*/作为注释结束,此种注释可以占用一行中的任意位置(当然不允许放在一个单词内部),也可以单独占用一行或多行。

下面是一个带注释的程序:

```
#include <iostream.h>    // 进行标准 I/O 操作需引入 iostream.h 头文件
#include <math.h>         // 使用数学函数需引入 math.h 头文件
/* 以下是主函数定义 */
void main()
{    // 向下为函数体
    double x,y,z;         // 定义三个实数变量
    x = 5;
    y = pow(x, 3);         // 计算  $x^3$ , 其值赋给 y
    z = sqrt(x);           // 计算  $\sqrt{x}$ , 其值赋给 z
    cout << x << ' ' << y << ' ' << z << endl;
}
```

该程序的运行结果如下:

```
5 125 2.23607
```

## 1.7 VC++ 6.0 集成开发环境简介

VC++ 6.0 是美国微软公司新近研制开发的 C++ 语言版本,它是一个集 C++ 程序编辑、编译、调试、运行和在线帮助等功能及可视化软件开发功能为一体的软件开发工具,或称开发环境、开发系统等。鉴于内容所限,本书只对其作简单介绍,目的是让读者掌握编辑、编译和运行一个 C++ 程序的简要过程。有兴趣的读者可参考该版本的使用说明和其他有关资料。

用 C++ 语言编写出一个完整的程序后,第一步需要上机输入和编辑程序,使之生成一个或多个程序文件以及一些头文件,其中有一个程序文件必含有主函数,被称之为程序主文件,简称主文件,它通常被首先输入和编辑;第二步对每个程序文件进行编译生成各自的目标代码(即二进制代码)文件,通常主文件被首先编译并生成主目标文件;第三步使主目标文件与同一程序中的其他目标代码文件以及有关 C++ 库函数文件相连接,生成一个可执行(即运行)文件;第四步运行最后生成的可执行文件,实现程序所具有的功能。



当由一个程序文件编译生成一个目标文件时,目标文件的主名与程序文件的主名相同,而扩展名改为 .obj。当由程序中的所有目标文件连接生成一个可执行文件时,该文件的扩展名为 .exe,主名为程序项目文件的主名,它通常与主文件的主名相同。

### 1. 输入和编译程序

用户在第一次进行VC++ 6.0集成开发环境之前最好先在某个逻辑磁盘上建立好一个用户目录,作为进入VC++ 6.0后使用的当前工作目录,用它来保存自己的头文件、程序文件、目标文件、运行文件以及系统自动生成的其他相关文件。

若在你所使用的机器上没有安装VC++ 6.0语言版本的软件,则应设法安装该软件,然后才能使用它。

若在 Windows 操作系统桌面上含有以 Microsoft Visual C++ 6.0 或 Msdev 为名字的图标,则双击后就启动该软件,在屏幕上打开VC++ 6.0集成开发环境操作界面窗口。否则,应单击屏幕左下角的“开始”按钮打开“开始”菜单,接着从中单击“程序”菜单项打开“程序”菜单。从中单击 Microsoft Visual Studio 6.0 菜单项打开该菜单。再接着单击或双击 Microsoft Visual C++ 6.0 菜单项运行该程序,就打开了相应的操作界面窗口,进入了VC++ 6.0集成开发环境。

打开的VC++ 6.0操作界面如图 1-1 所示,其中最顶行为窗口标题行,将显示出当前编辑的程序文件的文件名。第 2 行为菜单行,其中每一个菜单项都对应一个下拉式菜单,菜单中的每一个菜单项都是一条操作命令,都具有一定的操作功能。第 3 行为按钮工具行,向下左半部为工作区窗口,右半部为程序编辑窗口,整个操作界面的最下部为状态输出窗口。

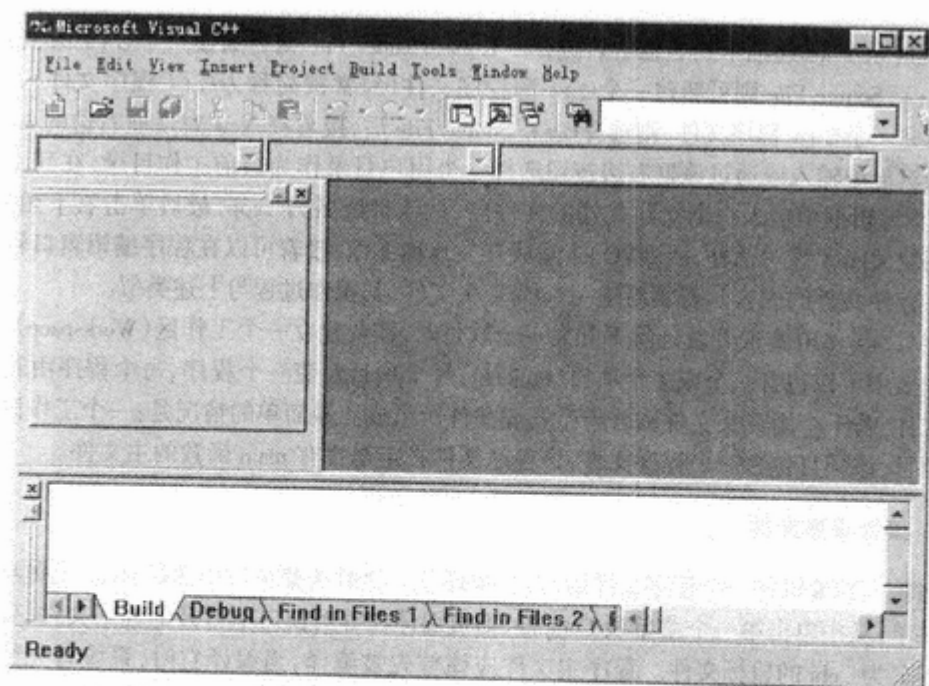


图 1-1 VC++ 6.0 操作界面窗口

为了新建或打开一个文件,请单击菜单栏中的 File 菜单打开此下拉菜单,若从中选择 New 菜单项则可以新建一个文件,若从中选择 Open 菜单项则可以打开一个已有的文件。当选择 New 菜单项后将打开 New 对话框,再选择 Files 标签则得到如图 1-2 所示。

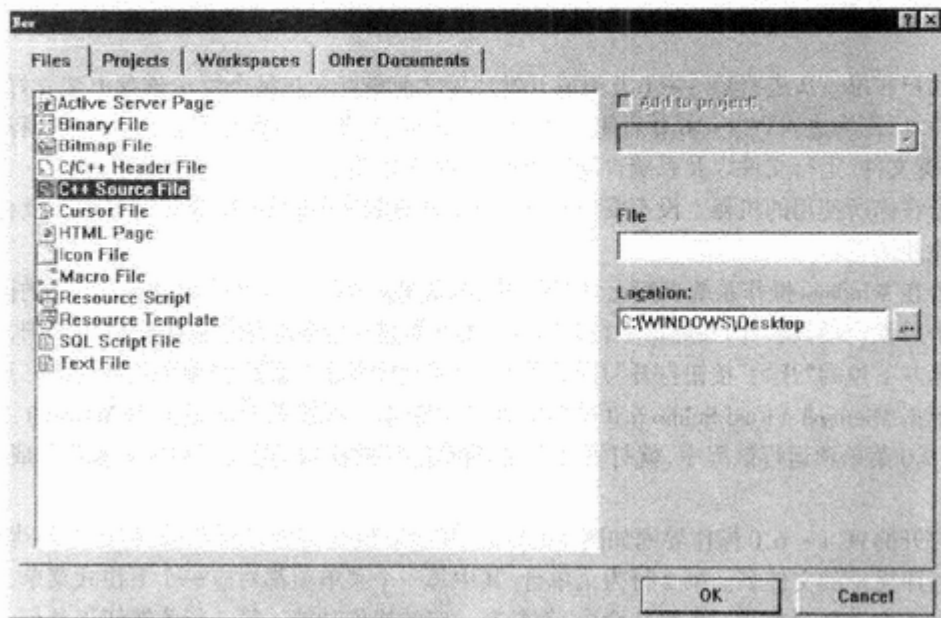


图 1-2 New 对话框

此时从 Files 选项卡列表框中若选择 C/C++ Header File 则可新建一个 C++ 头文件,若选择 C++ Source File 则可新建一个 C++ 源程序文件(经常被简称为 C++ 程序文件)。假定需要建立一个 C++ 程序文件,则选择 C++ Source File 后,接着在 New 对话框右边的 Location 文本编辑框中输入或通过该框右边按钮选择一个用户目录作为当前工作目录,在其上面的 File 文本编辑框中输入一个新建文件的文件名,默认扩展名为 .cpp,最后单击右下角的 OK 按钮则就关闭了该对话框,回到 VC++ 集成开发环境界面,接着可以在程序编辑窗口输入和编辑该程序文件的内容。若要新建一个 C++ 头文件,其操作过程与上述类似。

在 VC++ 6.0 集成开发环境下开发一个软件时,需要建立一个工作区(Workspace),在一个工作区中可以包含一个或多个项目(Project),每个项目对应一个程序,每个程序由若干个程序文件、若干个用户头文件和若干个资源文件所组成。最简单的情况是:一个工作区包含一个项目,该项目包含一个程序文件,该程序文件必定是含有 main 函数的主文件。

## 2. 编译程序文件

当输入和编辑好一个程序文件后,接着编译它。此时从菜单行中选择 Build 菜单项下拉出该菜单,从中单击第一个菜单项 Compile,即可编译在编辑窗口中打开的程序文件,生成一个扩展名为 .obj 的目标文件。程序主文件应该首先被编译,当编译它时,系统首先给出一个建立工作区的提示对话框,单击“是(Y)”按钮,表明同意在当前工作目录下建立一个工作区文件和一个项目文件,接着若文件没有存盘则系统自动给出一个提示存盘的对话框,单击

“是(Y)”按钮表示存盘。当编译一个非主程序文件时,若该文件没有被加入到主文件所在的项目中,则给出一个提示你把它加入到该项目的对话框,单击“是(Y)”按钮表示加入,然后系统才真正开始编译这个非主文件。

一个程序文件的编译过程是:首先处理所有预处理命令,如对于预处理包含命令,将它替换为所包含的头文件或程序文件的内容(使用 `# include` 命令也可以包含扩展名为 `.cpp` 的程序文件),接着删除掉所有注释内容(实际上是用一个空格取代),然后按行从上向下进行语法分析,最后生成相应的目标文件。

若在编译过程中检查出语法错误,则将在状态输出窗口显示出产生错误的程序行行号和错误原因,以便用户重新回到编辑窗口修改错误。

编译时检查出的错误包含两类:一类为严重错误(error),又称为致命错误,用户必须修改它,否则不能进一步向下处理;另一类为警告错误(warning),它不影响进入下一步处理过程,但最好把它修改掉,使得程序在编译后不产生任何错误。

当编译完一个程序文件后,将在状态输出窗口显示出各类错误的个数,若不出现任何错误则显示出“0 error(s), 0 warning(s)”信息,表示没有错误。

若你输入和编辑有多个程序文件和头文件,则它们的文件名将出现在菜单栏上的 Window 菜单项的下拉式菜单的底部,通过选择任一个文件名将使它成为当前的编辑文件,接着可对它进行编辑和编译。

注意:带扩展名为 `.h` 的头文件不能被编译。

### 3. 连接程序文件

连接程序文件就是将一个程序中的主目标文件与其他目标文件和相关的库函数目标文件连接起来形成一个可执行的文件。具体连接操作是:从菜单栏上单击 Build 菜单项下拉出相应菜单,接着从中单击第二个菜单项 Build 即可。若连接过程没有发现任何错误,则表示连接成功,此时在状态输出窗口显示出“0 error(s), 0 warning(s)”信息,若连接过程发现有错误,则将在状态输出窗口显示出发生错误的文件、所在的行号和出错原因,用户应根据这些信息修改有关程序文件中的错误,然后再重新进行编译和连接。

### 4. 运行程序

运行程序就是运行对该程序编译和连接而生成的可执行文件。具体操作是:从菜单栏上单击 Build 菜单项下拉出相应菜单,从中单击 Execute 菜单项即可。程序运行时将自动打开一个输出显示窗口,并且使显示光标处于该窗口左上角位置,每次执行程序中 `cout` 语句输出的内容和执行 `cin` 语句从键盘输入的内容都将从当前光标位置起显示出来,然后显示光标自动后移,即移到所显示内容的后面,再向显示器输出的内容将接着向后显示出来。程序在运行结束前,将在该窗口自动显示出“Press any key to continue”提示信息,用户按下任一键后将关闭该窗口,重新回到 VC++ 6.0 操作界面窗口。

## 习题一

### (一) 简答题

1. C++ 单词包含哪几种?
2. 一个标识符中的首字符必须是什么字符? 其余位置上的字符必须是什么字符?
3. 数值常量、字符常量、字符串常量和标识符在表示上各有什么区别?
4. 下面的每个数据各是什么常量、标识符、保留字、运算符、标点或非法数据?

25	8	3.48	'4'	"x1"	x2	"-28"
'd'	n	"y=m+1"	cout	cin	int	void
main	"a12.cpp"	if	"else"	endl	;"	+
'+'	"+"	Hlist	"int x;"	xy	A_1a	3ab

5. C++ 语句分为哪几类?
6. #include 命令的格式和功能各是什么? 使用尖括号和双引号在含义上有什么区别?
7. 你已经知道哪几个系统头文件?
8. cout 和 cin 标识符的含义是什么? 它们后面分别使用什么运算符? 各运算符的作用是什么?
9. 上机运行一个程序需要经过哪些阶段?

### (二) 填空题

1. 程序中的预处理命令是指以\_\_\_\_\_字符开头的命令。
2. 一条简单语句是以\_\_\_\_\_字符作为结束符的,一条复合语句是分别以\_\_\_\_\_字符和\_\_\_\_\_字符作为开始符和结束符的。
3. 空白符是\_\_\_\_\_符、\_\_\_\_\_符和\_\_\_\_\_符的统称。
4. 在 #include 命令中所包含的头文件,可以是\_\_\_\_\_头文件,也可以是\_\_\_\_\_头文件。
5. 使用 #include 命令可以包含一个头文件,也可以包含一个\_\_\_\_\_文件。
6. 一个函数定义由\_\_\_\_\_和\_\_\_\_\_两部分组成。
7. 一个程序中必须有一个名为\_\_\_\_\_的函数。
8. 函数头与\_\_\_\_\_语句一样都能够提供出函数的参数和返回类型等信息。
9. 若一个函数的定义处于调用它的函数之前,则在程序开始可以省去该函数的\_\_\_\_\_语句。
10. 一个函数的函数体就是一条\_\_\_\_\_。
11. C++ 头文件和源程序文件的扩展名分别为\_\_\_\_\_和\_\_\_\_\_。
12. 程序文件的编译错误分为\_\_\_\_\_和\_\_\_\_\_两类。
13. 当使用\_\_\_\_\_保留字作为函数类型时,该函数不返回任何值。
14. 当函数参数表用\_\_\_\_\_保留字表示时,则表示该参数表为空。
15. 从一条函数原型语句"int fun1(void);"可知,该函数的返回类型为\_\_\_\_\_,该函数
- 16.

带有\_\_\_\_\_个参数。

16. 当执行 cout 语句向 C++ 显示输出屏幕输出一个数据项的值时,若该值的显示宽度(即所占显示位置的字符数)为 n,则显示该值后光标将从原位置后移\_\_\_\_\_个字符显示位置。

17. 当执行 cout 语句输出 endl 数据项时,将使 C++ 显示输出屏幕上的光标从当前位置移动到\_\_\_\_\_的开始位置。

18. 当执行 cin 语句时,从键盘上输入每个数据后必须输入一个\_\_\_\_\_符,然后才能接着输入下一个数据。

### (三) 写出下列程序运行结果,此题又作为上机实验题

```
1. #include <iostream.h>
void main()
{
    int x,y;
    x=5; y=6;
    cout << "x+y=" << x+y << ',';
    cout << "x*y=" << x*y << endl;
}

2. #include <iostream.h>
int cube(int);
void main(void)
{
    cout << "cube(3) =" << cube(3) << endl;
    cout << "cube(5) =" << cube(5) << endl;
    cout << "cube(8) =" << cube(8) << endl;
}
int cube(int x)
{
    return x*x*x;
}

3. #include <iomanip.h>
#include "abc.cpp"
void main()
{
    double a,b,c;
    double averageValue;
    a=2;b=3;c=4;
    averageValue = AVE(a,b,c);
    cout << "averageValue:" << averageValue << endl;
    averageValue = AVE(a,b+1,c+2);
    cout << "averageValue:" << averageValue << endl;
}
```

其中 abc.cpp 文件的内容如下:

```
double AVE(double x, double y, double z)
{
```

```

        return (x+y+z)/3;
    }

4. #include <iostream.h>
   #include "example.h"
   void main()
   {
       int a,b,c;
       cout << "请输入 3 个整数:";
       cin >> a >> b >> c;
       cout << "最大值: " << max_value(a,b,c) << endl;
       cout << "最小值: " << min_value(a,b,c) << endl;
   }

```

其中 example.h 文件的内容如下:

```

int max_value(int a, int b, int c);
int min_value(int a, int b, int c);

```

这两个函数的定义(又称为函数的实现或具体实现)被保存在另一个程序文件中,它将被编译后连接到主文件中产生出可执行文件。该程序文件的内容如下:

```

int max_value(int a,int b, int c)
{
    if(a<b) a=b; // 若 a 小于 b 则将 b 的值赋给 a
    if(a<c) a=c; // 若 a 小于 c 则将 c 的值赋给 a
    return a;
}

int min_value(int a,int b, int c)
{
    if(a>b) a=b; // 若 a 大于 b 则将 b 的值赋给 a
    if(a>c) a=c; // 若 a 大于 c 则将 c 的值赋给 a
    return a;
}

```

请读者自行假定用于输入的 3 个整数。

## 第二章 数据类型和表达式

### 2.1 数据类型

数据是人们记录概念和事物的符号表示。如记录人的姓名用汉字表示,记录人的年龄用十进制数字表示,记录人的体重用十进制数字和小数点表示等,由此得到的姓名、年龄和体重都叫数据。根据数据的性质不同,可以把数据分为不同的类型。在日常使用中,数据主要被分为数值和文字(即非数值)两大类,数值又细分为整数和小数两类。

在C++语言中,数据分类如图2-1所示。

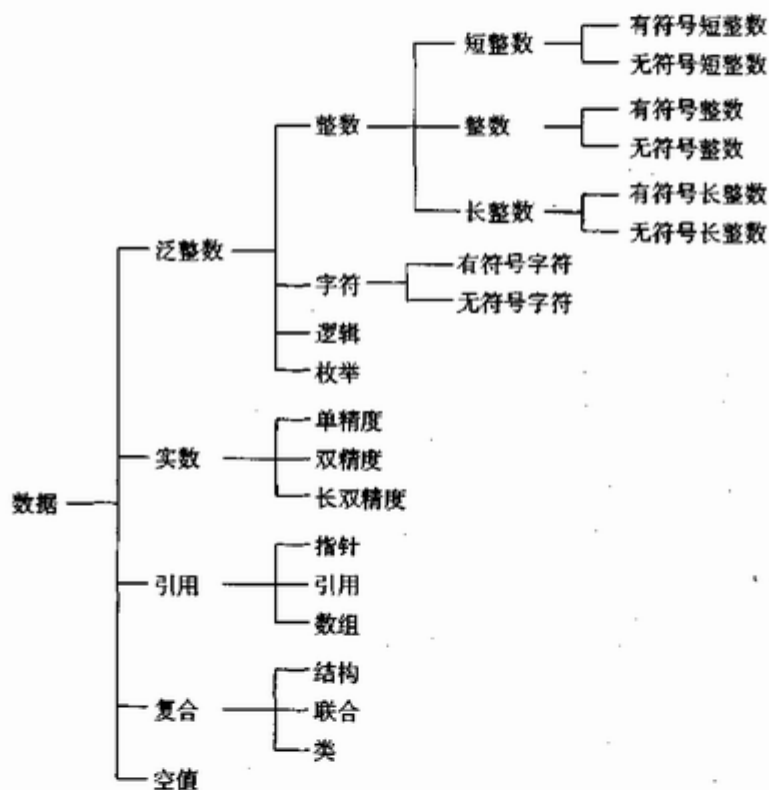


图 2-1 C++ 数据分类

图2-1中每一种无法再分解的数据类型为C++中的一种具体类型。每一种具体类型都对应着惟一的类型关键字、类型长度和值域范围,见表2-1所示,其中有些表项暂时空缺,留待以后介绍。



表 2-1 C++ 数据类型

类型	关键字	长度	值域范围
有符号短整数	short, short int, signed short int	2	$-2^{15} \sim 2^{15} - 1$ 内的整数
无符号短整数	unsigned short, unsigned short int	2	$0 \sim 2^{16} - 1$ 内的整数
有符号整数	int, signed int	4	$-2^{31} \sim 2^{31} - 1$ 内的整数
无符号整数	unsigned, unsigned int	4	$0 \sim 2^{32} - 1$ 内的整数
有符号长整数	long, long int, signed long int	4	$-2^{31} \sim 2^{31} - 1$ 内的整数
无符号长整数	unsigned long, unsigned long int	4	$0 \sim 2^{32} - 1$ 内的整数
有符号字符	char, signed char	1	$-128 \sim +127$ 内的整数
无符号字符	unsigned char	1	$0 \sim 255$ 内的整数
逻辑	bool	1	0 和 1
枚举	enum <枚举类型名>	4	为 int 值域内的一个子集
单精度数	float	4	$-3.402823 \times 10^{38} \sim 3.402823 \times 10^{38}$ 内的数
双精度数	double	8	$-1.7977 \times 10^{308} \sim 1.7977 \times 10^{308}$ 内的数
长双精度	long double	8	$-1.7977 \times 10^{308} \sim 1.7977 \times 10^{308}$ 内的数
指针	<类型关键字> *	4	$0 \sim 2^{32} - 1$ 内的整数
引用	<类型关键字> &		
数组	<类型关键字> [ <N> ]		
结构	struct <结构类型名>		
联合	union <联合类型名>		
类	class <类类型名>		
空值	void		

下面对表 2-1 作几点说明：

(1) 在每一种类型的关键字一栏中,用逗号分开的各组关键字是等价的,都是表示该类型的关键字。如 int 和 signed int 都表示有符号整数类型。

(2) 整数类型简称整型。大的整数类型包括小的整数类型、字符类型、逻辑类型和枚举类型,而小的整数类型又包括短整型(short int)、整型(int)和长整型(long int)这 3 种具体类型。读者应根据上下文联系来理解以后叙述中所用“整型”的含义。

(3) 对于每一种整数类型和字符类型,又可分为有符号和无符号两种类型。通常使用较多的是有符号类型,所以时常也把有符号类型简称为所属类型。如把有符号整数类型简称为整型或 int 型,把有符号字符类型简称为字符型或 char 型。

(4) 类型长度是指存储该类型值域范围内的任一个数据(又称为值)所占有的存储字节数,该字节数由系统规定,并且对任一数据都相同。如短整型长度为 2,即存储每个短整数占用两个字节,对应 16 个二进制位;整型长度为 4,即存储每个整数占用 4 个字节,对应 32 个二进制位;字符型长度为 1,即存储每个字符占用 1 个字节,对应 8 个二进制位。

(5) 类型的值域范围是指该类型所对应的固定大小的存储空间按照相应的存储格式所能表示的值的范围。如对于有符号短整型来说,它对应两个字节的存储空间,存储格式为二进制整数补码格式,只能够表示(即存储) $-2^{15} \sim 2^{15} - 1$ ,即  $-32768 \sim +32767$  之间的所有整数。若一个整数小于  $-32768$  或大于  $32767$ ,则它就不是该类型中的一个值,即它不是一个

短整数。又如对于无符号字符类型来说,它对应1个字节的存储空间,存储格式为二进制整数无符号(隐含为正)格式,只能够表示 $0 \sim 2^8 - 1$ ,即 $0 \sim 255$ 之间的所有整数。若一个整数小于0或大于255,则它就不是该类型中的一个值,即它不是一个字符数据。

(6) 一个数的有效数字是指从该数最左边不为0的数字位起至最右边不为0的数字位止之间的每一个数字位,而这些数字位的个数称为该数的有效数字位数。如3500,2.705,-0.278,63.00和0.00104的有效数字位数分别为2,4,3,2和3。另外,若一个数带有指数部分,则它不影响整个数的有效数字位数。如 $3.14$ , $3.14 \times 10^5$ , $314 \times 10^{-6}$ 等都具有相同的有效数字的位数,即都为3位。

(7) 单精度型的值域范围是从 $-3.402823 \times 10^{38} \sim 3.402823 \times 10^{38}$ 之间的不超过7位有效数字的所有整数和小数。如-372.65,  $-0.14 \times 10^{-6}$ , 0.0, +12.7, -6.45, 100.0,  $8.062 \times 10^{25}$ 等都是单精度范围内的数。2.00708463不是单精度范围内的一个数,若舍去它的最后两位有效数字,使之近似为2.007084,则就成为单精度范围内的一个数。

(8) 双精度型的值域范围比单精度型的值域范围更广,能够表示从 $-1.79769313486241 \times 10^{308} \sim 1.79769313486241 \times 10^{308}$ 之间的不超过15位有效数字的所有整数和小数。当一个数的有效数字的位数超过15时,则舍去第15位以后的所有位后,则可近似成为双精度范围内的一个数。在VC++中长双精度型与双精度型定义完全相同。

(9) 在VC++ 6.0版本中,整型(int)和长整型(long int)具有完全相同的长度和存储格式,所以它们是等同的。但在早期的C++版本中,由于当时的机器字长为16位,所以整型和长整型的长度是不同的,前者为两个字节,后者为4个字节。无论如何,任一种C++语言都遵循short int型的长度小于等于int型长度,同时int型长度又小于等于long int型长度的规定。

与上述情况类似,在VC++ 6.0中,双精度型(double)和长双精度型(long double)也具有完全相同的长度和存储格式,它们是等同的。在其他C++语言中也可能不同,但无论如何,它们都遵循float型的长度小于等于double型长度,同时double型长度又小于等于long double型长度的规定。

使用C++中的sizeof运算符(该运算符是一个保留字)能够很容易知道任一种数据类型的长度。如sizeof(int)的值就是一个整型的长度, sizeof(long double)的值就是一个长双精度型的长度。

(10) C++中的枚举、数组、结构、联合和类都是需要用户进行具体定义的类型,而其他所有类型都是预定义类型,即在C++系统内部已经定义了的类型。对于预定义类型,用户可以在程序中的任何地方直接使用它,对于用户定义类型,只有用户根据需要给出具体定义后,才能够在后面的程序中使用它。

## 2.2 常 量

常量是指在程序执行中不变的量,它分为字面常量和符号常量(又称标识符常量)两种表示方法。如25, -3.26, 'a', "constant"等都是字面常量,即字面本身就是它的值。符号常量是一个标识符,对应着一个存储空间,该空间中保存的数据就是该符号常量的值,这个数据

是在定义符号常量时赋予的,是以后不能改变的。如C++保留字中的 true 和 false 就是系统预先定义的两个符号常量,它们的值分别为数值 0 和 1。

关于符号常量的定义和赋初值的问题将在下一节同变量一起讨论,这一节只讨论字面常量的表示问题。

### 2.2.1 整型常量

整型常量简称整数,它有十进制、八进制和十六进制三种表示。

#### 1. 十进制整数

十进制整数由正号(+)或负号(-)开始的、接着为首位非 0 的若干个十进制数字所组成。若前缀为正号则为整数,若前缀为负号则为负数,若无符号则认为是正数。如 38, -25, +120, 74286 等都是符合书写规定的十进制整数。

当一个十进制整数大于等于  $-2147483648$  即  $-2^{31}$ ,同时小于等于  $2147483647$  即  $2^{31}-1$  时,则被系统看做是 int 型常量;当在  $2147483648 \sim 4294967295$  即  $2^{32}-1$  范围之内时,则被看做是 unsigned int 型常量;当超过上述两个范围时,则无法用C++整数类型表示,只有把它用实数(即带小数点的数)表示才能够被有效地存储和处理。

#### 2. 八进制整数

八进制整数由首位数字为 0 的后接若干个八进制数字(借用十进制数字中的 0~7)所组成。八进制整数不带符号位,隐含为正数。如 0,012,0377,04056 等都是八进制整数,对应的十进制整数依次为 0,10,255 和 2094。

当一个八进制整数大于等于 0 同时小于等于 01777777777 时,则称为 int 型常量,当大于等于 02000000000 同时小于等于 03777777777 时,则称为 unsigned int 型常量,超过上述两个范围的八进制整数则不要使用,因为没有相对应的C++整数类型。

#### 3. 十六进制整数

十六进制整数由数字 0 和字母 x(大、小写均可)开始的、后接若干个十六进制数字(0~9,A~F 或 a~f)所组成。同八进制整数一样,十六进制整数也均为正数。如 0x0,0x25,0x1ff,0x30CA 等都是十六进制整数,对应的十进制整数依次为 0,37,511 和 4298。

当一个十六进制整数大于等于 0 同时小于等于 0x7FFFFFFF 时,则称为 int 型常量,当大于等于 0x80000000 同时小于等于 0xFFFFFFFF 时,则称为 unsigned int 型常量,超过上述两个范围的十六进制整数没有相对应的C++整数类型,所以不能使用它们。

#### 4. 在整数末尾使用 u 和 l 字母

对于任一种进制的整数,若后缀有字母 u(大、小写等效),则硬性规定它为一个无符号整型(unsigned int)数,若后缀有字母 l(大、小写等效),则硬性规定它为一个长整型(long int)数。在一个整数的末尾,可以同时使用 u 和 l,并且对排列无要求。如 25U,0327UL,0x3ffL,648LU 等都是整数,其类型依次为 unsigned int,unsigned long int,long int 和 unsigned long int。

## 2.2.2 字符常量

字符常量简称字符,它以单引号作为起止标记,中间为一个或若干个字符。如 'a','%', '\n', '\012', '\125', '\x4F'等都是合乎规定的字符常量。每个字符常量只表示一个字符,当字符常量的一对单引号内多于一个字符时,则将按规定解释为一个字符。如 'a'表示字符 a, '\125'解释为字符 U(稍后便知是如何解释的)。

因为字符型的长度为 1,值域范围是 -128 ~ 127 或 0 ~ 255,而在计算机领域使用的 ASCII 字符,其 ASCII 码值为 0 ~ 127,正好在 C++ 字符型值域内。所以,每个 ASCII 字符均是一个字符型数据,即字符型中的一个值。

对于 ASCII 字符集中的每个可显示字符(个别字符除外),对应的 C++ 字符常量就是它本身,对应的值就是该字符的 ASCII 码,表示时用单引号括起来;对于像回车、换行那样的具有控制功能的字符,以及对于像单引号、双引号那样的作为特殊标记使用的字符,就无法采用上述的表示方法。为此引入了“转义”字符的概念,其含义是:以反斜线作引导的下一个字符失去了原来的含义,而转义为具有某种控制功能的字符。如 '\n'中的字符 n 通过前面使用的反斜线转义后就成为一个换行符,其 ASCII 码为 10。为了表示用做特殊标记使用的可显示字符,也需要用反斜线字符引导。如 '\''表示单引号字符,若直接使用''表示单引号是不行的,因为此时的单引号具有二义性。另外,还允许用反斜线引导一个具有 1~3 位的八进制整数或一个以字母 x 作为开始标记的具有 1 至 2 位的十六进制整数,对应的字符就是以这个整数作为 ASCII 码的字符。如 '\0', '\12', '\73', '\146', '\x5A'等对应的字符依次为空字符(其 ASCII 码为 0,注意:它不同于空格字符,空格字符的 ASCII 码为 32),换行符,':','f'和'Z'等。

由反斜线字符开始的符合上述使用规定的字符序列称为转义序列,C++ 语言中的所有转义序列如表 2-2 所示。

表 2-2 C++ 转义序列字符列表

转义序列	对应值	对应功能或字符	转义序列	对应值	对应功能或字符
\a	7	响铃	\\	92	反斜线
\b	8	退格	\'	39	单引号
\f	12	换页	\"	34	双引号
\n	10	换行	\?	63	问号
\r	13	回车	\ccc	ccc 的十进制值	该值对应的字符
\t	9	水平制表	\xhh	hh 的十进制值	该值对应的字符
\v	11	垂直制表			

转义序列不但可以作为字符常量,也可以同其他字符一样使用在字符串中。如 "abc \n"字符串中含有四个字符,最后一个为换行符, "\tx="中的首字符为水平制表符,当输出它时将使光标后移 8 个字符位置。

对于一个字符,当用于输出显示时,将显示出字符本身或体现出相应的控制功能,当出现在计算表达式中时,将使用它的 ASCII 码。如:

- (1) `char ch = 'E';`
- (2) `int x = ch + 2;`
- (3) `if(ch > 'C') cout << ch << ' ' << 'C' << endl;`
- (4) `cout << "apple \n";`

第一条语句定义字符变量 `ch` 并把字符 `E` 赋给它作为其初值,实际是把字符 `E` 的 ASCII 码 69 赋给 `ch`。第二条语句定义整型变量 `x` 并把 `ch + 2` 的值 71 赋给它。第三条语句首先进行 `ch > 'C'` 比较,实际上是取出各自的值(即对应的 ASCII 码)比较,因条件成立,所以执行其后的输出语句,将向屏幕输出 `E > C`。第四条语句输出一个字符串,即原样输出 `apple` 和使光标移到下一行开始位置。

### 2.2.3 逻辑常量

逻辑常量是逻辑类型中的值, `VC++` 用保留字 `bool` 表示逻辑类型,该类型只含有两个值,即整数 0 和 1,用 0 表示逻辑假,用 1 表示逻辑真。在 `VC++` 中还定义了这两个逻辑值所对应的符号常量 `false` 和 `true`, `false` 的值为 0,表示逻辑假, `true` 的值为 1,表示逻辑真。

由于逻辑值是整数 0 和 1,所以它也能够像其他整数一样出现在表达式里,参与各种整数运算。

### 2.2.4 枚举常量

枚举常量是枚举类型中的值,即枚举值。枚举类型是一种用户定义的类型,只有用户在程序中定义它后才能被使用。用户通常利用枚举类型定义程序中需要使用的一组相关的符号常量。枚举类型的定义格式为:

```
enum <枚举类型名> {<枚举表>;
```

这是一条枚举类型定义语句,该语句以 `enum` 保留字开始,接着为枚举类型名,它是用户命名的一个标识符,以后就直接使用它表示该类型,枚举类型名后为该类型的定义体,它是由一对花括号和其中的枚举表所组成,枚举表为一组用逗号分开的由用户命名的符号常量,每个符号常量又称为枚举常量或枚举值。如:

- (1) `enum color {red, yellow, blue};`
- (2) `enum day {Sun, Mon, Tues, Wed, Thur, Fri, Sat};`

第一条语句定义了一个枚举类型 `color`,用来表示颜色,它包含三个枚举值 `red`, `yellow` 和 `blue`,分别代表红色、黄色和蓝色。

第二条语句定义了一个枚举类型 `day`,用来表示日期,它包含 7 个枚举值,分别表示星期日、星期一至星期六。

一种枚举类型被定义后,可以像整型等预定义类型一样使用在允许出现数据类型的任何地方。如可以利用它定义变量。

- (1) `enum color c1, c2, c3;`
- (2) `enum day today, workday;`

(3) c1 = red;

(4) workday = Wed;

第一条语句开始的保留字 `enum` 和类型标识符 `colou` 表示上述定义的枚举类型 `color`, 其中 `enum` 可以省略不写, 后面的 3 个标识符 `c1`, `c2` 和 `c3` 表示该类型的 3 个变量, 每一个变量用来表示该枚举表中列出的任一个值。

第二条语句开始的两个成分(成分之间的空格除外)表示上述定义的枚举类型 `day`, 同样 `enum` 可以省略不写, 后面的两个标识符 `today` 和 `workday` 表示该类型的两个变量, 每一个变量用来表示该枚举表中列出的七个值中的任一个值。

第三条语句把枚举值 `red` 赋给变量 `c1`, 第四条语句把枚举值 `Wed` 赋给变量 `workday`。

在一个枚举类型的枚举表中列出的每一个枚举常量都对应着一个整数值, 该整数值可以由系统自动确认, 也可以由用户指定。若用户在枚举表中一个枚举常量后加上赋值号和一个整型常量, 则表示枚举常量被赋予了 this 整型常量的值。如:

```
enum day{Sun = 7, Mon = 0, Tues, Wed, Thur, Fri, Sat};
```

用户指定了 `Sun` 的值为 7, `Mon` 的值为 0。

若用户没有给一个枚举常量赋初值, 则系统给它赋予的值是它前一项枚举常量的值加 1, 若它本身就是首项, 则被自动赋予整数 0。如对于上述定义的 `color` 类型, `red`, `yellow` 和 `blue` 的值分别为 0, 1 和 2; 对于刚被修改定义的 `day` 类型, 各枚举常量的值依次为 7, 0, 1, 2, 3, 4, 5, 6。

由于各枚举常量的值是一个整数, 所以可把它同一般整数一样看待, 参与整数的各种运算。又由于它本身是一个符号常量, 所以当作为输出数据项时, 输出的是它的整数值, 而不是它的标识符, 这一点同输出其他类型的符号常量是一致的。

### 2.2.5 实型常量

实型常量简称实数, 它有十进制的定点和浮点两种表示方法, 不存在其他进制的表示。

#### 1. 定点表示

定点表示的实数简称定点数, 它是由一个符号(正号可以省略)后接若干个十进制数字和一个小数点所组成, 这个小数点可以处在任何一个数字位之前或之后。如: 12, 1.2, 12., 0.12, -12.40, +3.14, -.02037, -36.0 等都是符合书写规定的定点数。

#### 2. 浮点表示

浮点表示的实数简称浮点数, 它是由一个十进制整数或定点数后接一个字母 `e`(大、小写均可)和一个 1~3 位的十进制整数所组成, 字母 `e` 之前的部分称为该浮点数的尾数, 之后的部分成为该浮点数的指数, 该浮点数的值就是它的尾数乘以 10 的指数幂。如 3.23E5, +3.25e-8, 2E4, 0.376E-15, 1e-6, -6.04E+12, .43E0, 96.e24 等都是合乎规定的浮点数, 它们对应的数值分别为:  $3.25 \times 10^5$ ,  $3.25 \times 10^{-8}$ , 20000,  $0.376 \times 10^{-15}$ ,  $10^{-6}$ ,  $-6.04 \times 10^{12}$ , 0.43,  $96 \times 10^{24}$  等。

对于一个浮点数,若将它尾数中的小数点调整到最左边第一个非零数字的后面,则称它为规格化(或标准化)浮点数。如 21.6E8 和 -0.074E5 是非规格化的,若将它们分别调整为 2.16E9 和 -7.4E3 则都是规格化的浮点数。

### 3. 实数类型的确定

对于一个定点数或浮点数,C++ 自动按一个双精度数来存储,它占用 8 个字节的存储空间。若在一个定点数或浮点数之后加上字母 l(大、小写均可),则自动按一个单精度数来存储,它占用 4 个字节的存储空间。如 3.24 和 3.24f,虽然数值相同,但分别代表一个双精度数和一个单精度数,同样,-2.78E5 为一个双精度数,而 -2.78E5F 为一个单精度数。

#### 2.2.6 地址常量

指针类型的值域是  $0 \sim 2^{32} - 1$  之间的所有整数,每一个整数代表内存空间中一个对应单元(若存在的话)的存储地址,每一个整数地址都不允许用户直接使用来访问内存,以防止用户对内存系统数据的有意或无意的破坏。但用户可以直接使用整数 0 作为地址常量。它是 C++ 中惟一允许使用的地址常量,并称为空地址常量。它对应的符号常量为 NULL,表示不代表任何地址,在 iostream.h 等头文件中有此常量的定义。

## 2.3 变 量

变量是其值可以被改变的量。每一个变量都属于一种数据类型,用来表示(即存储)该类型中的一个值。在程序中只有存在了一种数据类型后,才能够利用它定义出该类型的变量。根据这一原则,我们可以随时利用 C++ 语言中的每一种预定义类型和用户已经定义的每一种类型定义所需要使用的变量。一个变量只有被定义后才能被使用,即才能进行存储和读取其值的操作。

### 1. 变量定义语句

变量定义是通过变量定义语句实现的,该语句的一般格式为:

<类型关键字> <变量名> [= <初值表达式>], ...;

<类型关键字> 为已存在的一种数据类型,如 short, int, long, char, bool, float, double 等都是类型关键字,分别代表系统预定义的短整型、整型、长整型、字符型、逻辑型(又称布尔型)、单精度型和双精度型。对于用户自定义的类型,可从类型关键字中省略其保留字。如假定 struct worker 是用户自定义的一种结构类型,则前面的保留字 struct 可以省略。

<变量名> 是用户定义的一个标识符,用来表示一个变量,该变量可以通过后面的可选项赋予一个值,称为给变量赋初值,<初值表达式> 是一个表达式,它的值就是赋予变量的初值。

该语句格式后面使用的省略号表示在一条语句中可以定义多个变量,但各变量定义之



间必须用逗号分开。

## 2. 语句格式举例

- (1) `int a, b;`
- (2) `char ch1 = 'a', ch2 = 'A';`
- (3) `int x = a + 2 * b;`
- (4) `double d1, d2 = 0.0, d3 = 3.14159;`

第一条语句定义了两个整型变量 `a` 和 `b`; 第二条语句定义了两个字符变量 `ch1` 和 `ch2`, 并被分别赋初值为字符 `a` 和 `A`; 第三条语句定义了一个整型变量 `x`, 并赋予表达式 `a + 2 * b` 的值作为初值; 第四条语句定义了 3 个双精度变量, 分别为 `d1`, `d2` 和 `d3`, 其中 `d2` 被赋予初值 `0.0`, `d3` 被赋予初值 `3.14159`。

## 3. 语句执行过程

当程序执行到一条变量定义语句时, 首先为所定义的每个变量在内存中分配与类型长度相同的存储单元, 如: 对每个整型变量分配 4 个字节的存储单元, 对每个双精度变量分配 8 个字节的存储单元; 接着若变量名后带有可选项, 则计算出初值表达式的值, 并把它保存到变量所对应的存储单元中, 表示给变量赋初值。若变量名后不带有可选项, 则当所属语句处于函数之外时, 将自动给变量赋予初值 `0`, 否则不赋予任何值, 此时的变量值是不确定的, 实际上是存储单元中的原有值(现在被称为垃圾)。

## 4. 语句应用举例

假定要计算一个圆的周长和面积, 则圆的半径、周长和面积都需要设定为变量, 假定分别用 `radius`, `girth` 和 `area` 标识符表示, 它们的类型均应为实数型, 即单精度或双精度型, 通常使用双精度型。根据圆的半径计算周长和面积的公式为:

```
girth = 2 * π * radius
area = π * radius * radius
```

下面给出用 C++ 语言编写的计算程序:

```
#include <iostream.h>
void main()
{
    double radius, girth, area;    // 定义变量
    cin >> radius;                // 从键盘输入一个圆的半径
    girth = 2 * 3.14159 * radius;  // 计算周长
    area = 3.14159 * radius * radius; // 计算面积
    cout << "radius: " << radius << endl;
    cout << "girth: " << girth << endl;
    cout << "area: " << area << endl;
}
```

在这个程序的主函数中, 第一条语句定义了 3 个变量, 由于没有给它们赋初值, 所以其值是不确定的; 第二条语句从键盘输入一个常数给半径 `radius`, 输入的常数可以是整数, 也可

以是定点数或浮点数,系统将自动把它转换为一个双精度数后再赋给 `radius`,即赋给该变量所对应的存储单元;第三条和第四条语句分别计算出赋值号右边表达式的值,再分别赋给变量 `girth` 和 `area`;第五至七条语句依次向屏幕输出圆的半径、周长和面积。

假定程序运行后从键盘上输入的半径为 5.62,则得到的输出结果为:

```
radius:5.62
girth: 35.3115
area: 99.2252
```

## 5. 符号常量定义语句

符号常量定义语句同变量定义语句类似,其语句格式为:

```
const <类型关键字> <符号常量名> = <初值表达式>, ...;
```

该语句以保留字 `const` 开始并标识,后跟符号常量的类型关键字,接下去为符号常量名,它是一个用户定义的标识符,符号常量名之后为一个赋值号和一个表达式(注意:表达式中既可以含有常量也可以含有变量),由此可见,在定义符号常量时必须同时对其赋初值。该语句同样也可以定义多个符号常量。

系统执行符号常量定义语句也同执行变量定义语句一样,需要依次为每个符号常量分配存储单元并赋初值。

一个符号常量被定义后,它的值就是定义时所赋予的初值,以后将始终保持不变,因为系统只允许读取它的值,而不允许向它赋值。

另外,在符号常量的定义语句中,若 `<类型关键字>` 为 `int`,则可以被省略。

下面给出几个符号常量定义语句的例子:

- (1) `const int A1 = 5, A2 = A1 * 4;`
- (2) `const double PI = 3.14159;`
- (3) `const int MaxSize = 100;`

第一条语句定义了两个整型符号常量 `A1` 和 `A2`,并使得它们的初值分别为 5 和 20;第二条语句定义了一个双精度符号常量 `PI`,用它表示数学上  $\pi$  的值 3.14159;第三条语句定义了一个整型符号常量 `MaxSize`,用它代表整数 100。第一条和第三条语句中的 `int` 均可以省略不写。

符号常量定义语句既可以出现在函数体外,也可以出现在函数体内,这一点也跟变量定义语句相同。

符号常量也必须遵循先定义后使用的原则,这也与变量的定义和使用的规则相同。

## 6. 使用 `#define` 命令定义符号常量

`#define` 命令是一条预处理命令,其命令格式为:

```
#define <符号常量名> <字符序列>
```

`<符号常量名>` 是用户定义的标识符,又称为宏或宏标识符,`<字符序列>` 也是由用户给定的、将用来代替宏的一个字符序列。宏被该命令定义后,可以使用在其后的程序中,当程序被编译时将把所有地方使用的宏标识符替换为对应的 `<字符序列>`,并把宏命令删

除掉。

如一个宏命令为：

```
#define ABC 10
```

若在主函数中有这样一条语句：

```
int x = ABC * ABC;
```

则当编译后改变为：

```
int x = 10 * 10;
```

若上述宏命令中的字符序列不是 10, 而是 2+5, 则编译后改变为：

```
int x = 2 + 5 * 2 + 5;
```

可见宏替换后改变了原表达式中运算的优先次序, 为了克服可能出现的这种错误, 通常使用带括号的宏字符序列。如可将上述定义的宏命令改写为：

```
#define ABC (2 + 5)
```

上述语句将会被正确地替换为：

```
int x = (2 + 5) * (2 + 5);
```

由于使用 const 语句定义符号常量带有数据类型, 以便系统进行类型检查, 同时该语句具有计算初值表达式和给符号常量赋初值的功能, 所以使用它比使用宏命令定义符号常量要优越得多, 因此提倡在程序中使用 const 语句定义符号常量。

## 7. 使用变量和常量的程序举例

```
#include <iostream.h>
#define M -1 // 符号常量中的字母通常采用大写
const int N = 10;
void main()
{
    int x, y;
    cout << "请输入一个整数:";
    cin >> x;
    if (x < N) y = M * x + 1;
    else y = (x + M) * x - 3;
    cout << x << " " << y << endl;
}
```

程序运行后若从键盘上输入数值 5, 则得到的输出结果为：

```
5 -4
```

若从键盘上输入的数值为 20, 则得到的输出结果为：

```
20 377
```

## 2.4 运算符

C++ 运算符又称操作符,它是对数据进行运算的符号,参与运算的数据称为操作数或运算对象,由操作数和操作符连接而成的有效的式子称为表达式。

按照运算符要求操作数个数的多少,可把C++ 运算符分为单目(或一元)运算符、双目(或二元)运算符和三目(或三元)运算符三类。单目运算符一般位于操作数的前面,如对  $x$  取负为  $-x$ ;双目运算符一般位于两个操作数之间,如两个数  $a$  和  $b$  相加表示为  $a+b$ ;三目运算符只有一个,即为条件运算符,它含有两个字符,分别把三个操作数分开。

一个运算符可能是一个字符,也可能由两个或三个字符所组成,还有的是一些C++ 保留字。如赋值号( $=$ )就是一个字符,不等于号( $!=$ )就是两个字符,左移赋值号( $<<=$ )就是三个字符,测类型长度运算符( $sizeof$ )就是一个保留字。

每一种运算符都具有一定的优先级,用来决定它在表达式中的运算次序。一个表达式中通常包含有多个运算符,对它们进行运算的次序通常与每一个运算符从左到右出现的次序相一致,但若它的下一个(即右边一个)运算符的优先级较高,则下一个运算符应先被计算。如当计算表达式  $a+b*(c-d)/e$  时,则每个运算符的运算次序依次为:  $-$ ,  $*$ ,  $/$ ,  $+$ 。

对于同一优先级的运算符,当在同一个表达式的计算过程中相邻出现时,可能是按照从左到右的次序进行,也可能是按照从右到左的次序进行,这要看运算符的结合性。如加和减运算为同一优先级,它们的结合性是从左到右,当计算  $a+b-c+d$  时,先做最左边的加法,再做中间的减法,最后做右边的加法;又如各种赋值操作是属于同一优先级,但结合性是从右到左,当计算  $a=b=c$  时,先做右边的赋值,使  $c$  的值赋给  $b$ ,再做左边的赋值,使  $b$  的值赋给  $a$ 。

表 2-3 列出了在C++ 语言中定义的全部运算符,其中优先级数字从小到大对应的优先级别为从高到低。

表 2-3 C++ 运算符

优先级	运算符	功 能	目 数	结合性
1	::	作用域区分符	双目	从左向右
	()	改变运算优先级或函数调用操作符		
	[]	访问数组元素		
	.	直接访问数据成员		
	->	间接访问数据成员		
2	!	逻辑非	单目	从右向左
	~	按位取反		
	+, -	取正,取负		
	*	间接访问对象		
	&	取对象地址		
	++, --	增 1, 减 1		
	()	强制类型转换		
	sizeof	测类型长度		
	new	动态申请内存单元		
	delete	释放 new 申请的单元		

(续表)

优先级	运算符	功 能	目 数	结合性
3	.	引用指向类成员的指针	双目	从左到右
	-> *	引用指向类成员的指针		
4	*, /, %	乘, 除, 取余	双目	从左向右
5	+, -	加, 减		
6	<<, >>	按位左移, 按位右移		
7	<, <=, >, >=	小于, 小于等于, 大于, 大于等于		
8	==, !=	等于, 不等于		
9	&	按位与		
10	^	按位异或		
11		按位或		
12	&&	逻辑与		
13		逻辑或		
14	?:	条件运算符	三目	从右向左
15	=	赋值	双目	从右向左
	+=, -=	加赋值, 减赋值		
	*=, /=	乘赋值, 除赋值		
	%=, &=	取余赋值, 按位与赋值		
	^=	按位异或赋值		
	=	按位或赋值		
	<<=	按位左移赋值		
	>>=	按位右移赋值		
16	,	逗号运算符	双目	从左向右

下面对表 2-3 中的一些运算符作简要介绍, 对剩余的一些运算符将结合以后各章的有关内容一同介绍。

### 1. 双目算术运算符

这类运算符包括加、减、乘、除和取余等 5 种, 它们的含义与数学上相同。该类运算的操作数为任一种数值类型, 包括任一种整数类型和任一种实数类型。由算术运算符(包括单目和双目)连接操作数而成的式子称为算术(或数值)表达式, 每个算术表达式的值为一个数值, 其类型按如下规则确定。

当参加运算的两个操作数均为整型(但具体类型可以不同, 如一个为 `int` 型, 另一个为 `char` 型)时, 则运算结果为 `int` 型(因在 `VC++` 中 `int` 和 `long int` 的值域范围相同, 所以均可认为是 `int` 型)。

注意: 两个整数相除得到的是它们的整数商, 两个整数取余得到的是整余数。

当参加运算的两个操作数中至少有一个是单精度型, 并且另一个不是双精度型时, 则运算结果为 `float` 型。

当参加运算的两个操作数中至少有一个是双精度型时, 则运算结果为 `double` 型。

假定整型变量 `x` 和 `y` 的值分别为 25 和 6, 则下面给出整数运算, 特别是含有除和取余运

算的例子:

```
x/8 = 3          x/y + 5 = 9
10 - y % x = 4   x % 5 = 0
x * 3 % 4 = 3    65 % x / 3 = 5
-56 / 6 = -9     -56 % 6 = -2
```

若要使两个整数相除得到一个实数,则必须将其中之一转变为实数。如:

```
9.0/2 = 4.5      -15/4.0 = -3.75
float(y)/x = 0.24 x/double(-8) = -3.125
```

其中 `float(y)` 和 `double(-8)` 分别表示把括号内的表达式的值分别转换为一个单精度数和双精度数。

## 2. 赋值运算符

赋值运算除了一般的赋值运算(=)外,还包括各种复合赋值运算,如 +=, -=, \*=, /= 等。一般赋值运算采用的赋值号借用数学上的等号,其功能是把赋值号右边的表达式的值赋给左边变量所对应的存储单元中。由一般或复合赋值号连接左边变量和右边表达式而构成的式子称为赋值表达式,每个赋值表达式都有一个值,它就是通过赋值得到的左边变量的值。如 `x = 3 * 15 - 2` 的值就是通过赋值保存在 `x` 中的值 43。

通常在一个赋值表达式中,赋值号两边的数据类型是相同的,若出现不同时,则在赋值前自动会把右边表达式的值转换为与左边变量类型相同的值,然后再把这个值赋给左边变量。如执行 `x = 20/3.0` 时,若 `x` 为整型,则得到的 `x` 的值为 6,它是将右边计算得到的双精度值舍去小数部分,只保留整数部分 6 的结果。再如,执行 `y = 40` 时,若 `y` 为双精度变量,则首先把 40 转换为双精度数 40.0(或表示为 4.0e1)后再赋给 `y`。

注意:当把一个实数值赋给一个整型量时,将丢失小数部分,获得的只是整数部分,它是实数的一个近似值。

在一个赋值表达式中可以使用多个赋值号实现给多个变量赋值的功能。如执行 `x = y = z = 0` 时就能够同时给 `x`, `y` 和 `z` 赋值 0。由于赋值号的结合性是从右向左,所以实际赋值过程是:首先把 0 赋给 `z`,得到子表达式 `z = 0` 的值为 `z` 的值 0,接着把这个值赋给 `y`,得到子表达式 `y = z = 0` 的值为 `y` 的值 0,最后把这个值赋给 `x`,使 `x` 的值也为 0。整个表达式的值也就是 `x` 的值 0。

赋值号也可以使用在常量和变量的定义语句中,用于给它们赋初值。但这里的赋值号只起到赋初值的作用,并不构成赋值表达式。

在 C++ 中有许多复合赋值运算符,每个运算符的含义为:把右边表达式的值同左边变量的值进行相应运算后,再把这个运算结果赋给左边的变量,该复合赋值表达式的值也就是保存在左边变量中的值。如执行 `x += 3` 时,就是把 `x` 的值加上 3 后再赋给 `x`,它与执行 `x = x + 3` 表达式的计算是等价的,若 `x` 的值为 5,则计算后得到的 `x` 的值为 8,它也是这个表达式的值。

对于任一种赋值运算,其赋值号或复合赋值号左边必须是一个左值。左值是指具有对应的可由用户访问的存储单元,并且能够由用户改变其值的量。如一个变量就是一个左值,因为它对应着一个存储单元,并可由编程者通过变量名访问和改变其值。而一个字面常量

就不是一个左值,因为它不存在供用户访问并能改变其值的存储单元。一个通过 `const` 语句定义的符号常量也不是一个左值,因为虽然用户能访问它,但不能改变它的值。一般的算术表达式(如  $x * 5 + 2$ )也不是一个左值,因为它的值只能随时被使用,不能再访问和改变它。由此可知:表达式  $(x + 5) = 10$  是非法的,因为赋值号左边的  $(x + 5)$  是一个值,而不是一个左值,常量 10 无法赋给它。不是左值的量被称为右值。

一个赋值表达式的结果实际上是一个左值,指的是赋值号左边的变量。如  $x = y - 2$  的值就是被赋值后的  $x$ ,它是一个左值,代表着对应存储单元中的值。同样,  $x * = y$  的结果也是一个左值,即  $x$ 。表达式  $(x += 5) * = 2$  是合法的,其结果为左值  $x$ ,若  $x$  的原值为 5,则最后得到的  $x$  值为 20。

### 3. 增 1 和减 1 运算符

增 1 运算符用连续两个加号 (`++`) 表示,减 1 运算符用连续两个减号 (`--`) 表示。它们都是单目操作符,并且要求操作数必须是左值,通常为一个变量,操作数的类型可以是任一种整数类型,不过对于枚举类型需要有相应操作符重载的定义。

`++` 和 `--` 运算符有两种使用格式:一是使用在操作数的前面,另一是使用在操作数的后面,它们都是将操作数分别增 1 或减 1,但含义略有区别。进行 `++` 或 `--` 运算构成的表达式称为增 1 或减 1 表达式,当操作符使用在前面时,首先使操作数增 1 或减 1,然后求出表达式的值就是被增 1 或减 1 后的操作数,它是一个左值;当操作符使用在后面时,同样使操作数增 1 或减 1,但求出的表达式的值是运算前的操作数的值,注意:它不是一个左值。

假定下面每个表达式中整型变量  $x$  的值均为 10,则:

- (1) `++x`           // 表达式的值为增 1 后的  $x$ ,值为 11
- (2) `x++`           //  $x$  变为 11,但表达式的值为 10
- (3) `--x`           // 表达式的值为减 1 后的  $x$ ,值为 9
- (4) `x--`           //  $x$  变为 9,但表达式的值为 10
- (5) `++x = 5`       //  $x$  首先变为 11,然后变为 5,此语句合法,但可能没意义
- (6) `x++ = 5`       // `x++` 不是左值,因此不能被赋值,此表达式非法
- (7) `--x--`         // 因 `--` 的结合性为从右向左,所以先得到  $x$  的值为 9,  
                      // `x--` 的值为 10,它不是左值,当接着进行左边减 1 时非法
- (8) `--++x`         // 结果仍为左值  $x$
- (9) `y = x++`        //  $x$  变为 11, $y$  的值为 10
- (10) `y = --x`       //  $x$  变为 9, $y$  的值为 9
- (11) `y = 5 * x++`   //  $x$  变为 11, $y$  的值为 50
- (12) `y = x * ++x;` //  $y$  的值为 121。当算术表达式中多处出现有同一变量时,  
                      // 最好不要对它进行 `++` 或 `--` 操作,以免发生混乱。如应将  
                      // 此语句改写为“`x++;`”和“`y = x * x;`”这两条语句

还要注意:`x++` 和  $x + 1$  是不同的表达式,`x++` 的值为  $x$  的原值, $x$  的值为增 1 后的值, $x + 1$  的值为  $x$  的值加 1 后的结果,运算前后  $x$  的值不变。`++x` 和  $x += 1$  及  $x = x + 1$  的作用是完全相同的。同理,`x--` 与  $x - 1$  不同,而 `--x` 和  $x -= 1$  及  $x = x - 1$  的作用是完全相同的。



#### 4. 测类型长度运算符

该运算符的使用格式为:

`sizeof(<类型名或表达式>)`

运算结果是类型名所表示类型的长度或表达式的值所占用的字节数,亦即这个值所属类型的长度。如:

- (1) `sizeof(int) = 4`
- (2) `sizeof(double) = 8`
- (3) `sizeof(100) = 4`
- (4) `sizeof('a') = 1`
- (5) `sizeof(struct ABC)`      // 求出结构类型 ABC 的长度
- (6) `sizeof(a)`      // 求出变量 a 的长度,亦即它所占用的字节数

#### 5. 强制类型转换

强制类型转换是把一种类型的数据转换为另一种类型的数据,转换格式为:

`<类型关键字>(<表达式>)`

或:

`(<类型关键字>)<表达式>`

或:

`(<类型关键字>)(<表达式>)`

无论待转换的<表达式>是什么形式的数据,如常量、变量、函数调用或带有运算符的表达式,则转换后得到的是<类型关键字>所属类型的一个值,它不是一个左值。假定 x 为 int 型,其值为 80, r 为字符型,其值为 'd', 对应的 ASCII 值为 100, 则:

- (1) `float(x) = 80.0`      // 结果为 float 型,当然 x 的类型和值不变
- (2) `double(-1) = -1.0`      // 结果为 double 型
- (3) `char(x) = 'P'`      // 结果为 char 型, x 的类型和值不变
- (4) `int(r) = 100`      // 结果为 int 型, r 的类型和值不变
- (5) `(double)'h' = 104.0`      // 结果为 double 型
- (6) `(int *)p`      // 把 p 的值转换为一个整数类型的指针值
- (7) `(short int)(5 * x - 6)`      // 把  $5 * x - 6$  的值转换为一个短整型值
- (8) `(char *)(p++)`      // 把 p 的原值转换为一个字符型指针值,同时 p 增 1

#### 6. 按位操作符

按位操作符要求操作数必须是整型、字符型和逻辑型数据。一个数按位左移(<<)多少位通常将使结果比操作数扩大了 2 的多少次幂;按位右移(>>)多少位将通常使结果比操作数缩小了 2 的多少次幂;按位取反(~)使结果为操作数的按位反,即 0 变 1 和 1 变 0;按

位与(&)使结果为两个操作数的对应二进制位的与,1和1的与得1,否则为0;按位或(|)使结果为两个操作数的对应二进制位的或,0和0的或得0,否则为1;按位异或(^)使结果为两个操作数的对应二进制位的异或,0和1及1和0的异或得1,否则为0。每一种按位操作的结果都是一个值,但不是左值。

假定若整数变量  $x = 24$ ,  $y = 36$ , 它们对应的二进制表示为 00011000 和 00100100, 因高位三个字节的价值均为 0, 所以省略不写, 则:

- (1)  $x \ll 2 = 96$  //  $x$  的值不变, 表达式值为对  $x$  值左移 2 位而得到的值 96
- (2)  $y \gg 3 = 4$  //  $y$  的值不变, 表达式值为 4, 约为  $y$  的  $1/8$
- (3)  $\sim x = -25$  //  $x$  不变, 表达式的值为 -25
- (4)  $x \& y = 0$  //  $x$  和  $y$  不变, 表达式的值为 0
- (5)  $x | y = 60$  //  $x$  和  $y$  不变, 表达式的值为 60
- (6)  $x | 44 = 60$  // 表达式的值为 60
- (7)  $x \wedge 44 = 52$
- (8)  $x | x \ll 2 = 120$  //  $x$  的值不变, 表达式的值等于 120

## 7. 关系运算符

关系运算符共有六个: 小于(<)、小于等于(<=)、大于(>)、大于等于(>=)、等于(==)和不等(!=), 它们都是双目运算符, 用来比较两个操作数的大小, 运算结果均为逻辑值 0 或 1。由一个关系运算符连接前后两个数值表达式而构成的式子称为关系表达式, 简称关系式。当一个关系式成立时则计算结果为逻辑值真(1), 否则为逻辑值假(0)。

假定  $x = 20$ ,  $y = 3.25$ ,  $ch$  为一个字符变量, 则:

- (1)  $x == 0$  // 不成立, 结果为 0
- (2)  $x != y$  // 成立, 结果为 1
- (3)  $x++ > = 21$  // 不成立, 结果为 0,  $x$  变为 21
- (4)  $++x == 21$  // 成立, 结果为 1,  $x$  变为 21
- (5)  $y + 10 < y * 10$  // 成立, 结果为 1
- (6)  $x-- < 20$  // 不成立, 结果为 0,  $x$  变为 19
- (7)  $'a' == 'A'$  // 不成立, 结果为 0
- (8)  $ch! = 0$  // 或写成  $ch! = '\0'$ , 当  $ch$  非 0 时结果为 1, 否则为 0

在上述六个关系运算符中, 可分为三组: < 和 >=、> 和 <=、== 和 !=, 每组中的两个运算符是互为反运算, 当一种运算结果为 1 时, 它的反运算结果必然为 0。如当  $x > y$  成立时, 其逻辑值为 1, 它的反运算  $x <= y$  必然不成立, 其逻辑值为 0。由反运算构成的式子称为原式的相反式。如  $x > y$  和  $x <= y$  是互为相反式。

## 8. 逻辑运算符

逻辑运算符有三个: 逻辑非(!)、逻辑与(&&)和逻辑或(||), 其中! 为单目运算符, && 和 || 为双目运算符。逻辑运算的对象是逻辑值 0 或 1, 若它不是一个逻辑值, 则对于非 0 值首先转换为逻辑值 1, 对于 0 值转换为逻辑值 0。逻辑运算的结果是一个逻辑值 0 或 1。

逻辑非是对操作对象取反, 若操作对象为 1, 则运算结果为 0, 若操作对象为 0, 则运算结

果为1;逻辑与的结果是当两个操作对象都为1时,其值为1,否则为0;逻辑或的结果是当两个操作对象都为0时,其值为0,否则只要有一个为1其结果为1。逻辑非、与、或的运算规则如表2-4所示。

表 2-4 逻辑运算规则

a	b	! a	a && b	a    b
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

逻辑运算的操作数是逻辑型数据,逻辑常量、逻辑变量、关系表达式等都是逻辑型数据,由逻辑型数据和逻辑运算符连接而成的式子称为逻辑表达式,简称逻辑式。一个数值表达式也可以作为一个逻辑型数据使用,当值为0时则认为是逻辑值0,当值为非0时则认为是逻辑值1。总之,任何一个具有0和非0取值的式子都可以作为逻辑表达式使用。

仍假定  $x=20, y=3.25$ , 则:

- (1)  $x > 0 \ \&\& \ y > 0$       // 1&&1, 结果为1
- (2)  $x > 0 \ \&\& \ \text{true}$       // 1&&1, 结果为1
- (3)  $x \ \&\& \ \text{false}$       // 1&&0, 结果为0
- (4)  $!x == 0$       // 结果为1, 注意: 先进行非运算, 后进行等于运算
- (5)  $!(x > 0)$       // 结果为0
- (6)  $! \ x \ || \ y < 1$       // 结果为0
- (7)  $x < -10 \ || \ x > 10$       // 0||1, 结果为1
- (8)  $x++! = 20 \ || \ y$       // 0||1, 结果为1
- (9)  $x <= 0 \ \&\& \ x < y$       // 0&&0, 结果为0

在逻辑运算中,存在着以下三种等价关系:

- (1)  $!!a == a$
- (2)  $!(a \ \&\& \ b) == !a \ || \ !b$
- (3)  $!(a \ || \ b) == !a \ \&\& \ !b$

其中a和b均代表逻辑量,即取值为逻辑值0和1的量。这三个等价关系的含义为:对一个逻辑量的两次取反仍等于它本身;对两个逻辑量的先与再取反等于分别对它们取反后再或;对两个逻辑量的先或再取反等于分别对它们取反后再与。这可以从表2-5得到证明。

表 2-5 证明逻辑等价关系的真值表

a	b	!! a	! (a && b)	! a    ! b	! (a    b)	! a && ! b
0	0	0	1	1	1	1
0	1	0	1	1	0	0
1	0	1	1	1	0	0
1	1	1	0	0	0	0

由表2-5可以看出:上面每一种等价关系是成立的,因为对于a和b的任意组合,两边

的逻辑值是相等的。

一个逻辑表达式的逻辑非称为这个逻辑表达式的相反式,根据上述的等价关系,很容易求出一个逻辑表达式的相反式。如:

- |                                      |                                    |
|--------------------------------------|------------------------------------|
| (1) $x \geq 5$                       | $x < 5$                            |
| (2) $x$                              | $!x$                               |
| (3) $x == 0 \ \&\& \ y > 1$          | $x != 0 \    \ y \leq 1$           |
| (4) $x > 1 \ \&\& \ x \leq 20$       | $x \leq 1 \    \ x > 20$           |
| (5) $x! = key \ \&\& \ flag == true$ | $x == key \    \ flag != false$    |
| (6) $ch < 'a' \    \ ch > 'z'$       | $ch \geq 'a' \ \&\& \ ch \leq 'z'$ |
| (7) $a == b \    \ a == c$           | $a! = b \ \&\& \ a! = c$           |
| (8) $a \geq x \    \ b > 2 * y + 10$ | $a < x \ \&\& \ b \leq 2 * y + 10$ |

从每个例子可以看出:一个逻辑表达式和它的相反式是互为相反式。能够求出一个逻辑表达式的相反式,对于提高程序分析和设计能力很有帮助。

### 9. 条件运算符

条件运算符(?)是C++中惟一一个三目运算符,其使用格式为:

<表达式1> ? <表达式2> : <表达式3>

当计算由条件运算符构成的表达式时,首先计算<表达式1>,若其值非0则计算出<表达式2>的值,这个值就是整个表达式的值;若<表达式1>的值为0,则计算出<表达式3>的值,它就是整个表达式的值。如:

- (1)  $a = (x > y ? x : y)$       // 若  $x > y$  为真则把  $x$  的值赋给  $a$ , 否则把  $y$  的值赋给  $a$   
(2)  $x ? y = a + 10 : y = 3 * a - 1$       // 若  $x$  非0则把  $a + 10$  的值赋给  $y$ , 否则把  $3 * a - 1$  的值赋给  $y$

### 10. 逗号运算符

逗号运算符是一种顺序运算符,对于分别用逗号分开的若干个表达式,每个逗号都称为逗号运算符,合起来称为逗号表达式。计算一个逗号表达式时,将按照每个子表达式从左到右出现的先后次序依次计算出它们的值,最后一个子表达式的值就是整个表达式的值。如  $(x++, y += x, z--)$  就是一个逗号表达式,它首先计算  $x++$  的值,该计算使  $x$  增1;接着计算  $y += x$  的值,该计算使  $y$  增加了  $x$  的值;最后计算  $z--$  的值,使  $z$  减1,而  $z$  的原值则成为整个表达式的值。

### 11. 圆括号运算符

使用圆括号能够改变运算的优先级,使得括号内的运算优先进行,这与数学上的含义相同。

在C++语言中,运算符比较多,级别划分得也比较细,往往不容易正确地记住每个运算符的优先级,因此也就不容易把它们正确地使用在复杂的表达式中。为了使表达式中每个运算符的运算次序能够按照你所希望的次序进行,请使用圆括号进行限定,即使有时是多余的,也没有关系,因为它还能够使表达式更清晰,提高程序的可读性。如:

- (1)  $x > 0 \ \&\& \ x < 3$       // 表示为  $(x > 0) \ \&\& \ (x < 3)$  可能更清晰

(2) `cout << (x > y ? x : y) << endl;`

在第二条语句中,若不使用括号是错误的,因为 `<<` 高于 `>` 和 `?:` 的优先级,就不能把条件表达式作为一个整体看待了。

**注意:**在 `cout` 语句中, `<<` 不是左移操作符,而是重新赋予了把其后的一个数据项的值插入(即输出)到屏幕输出窗口的含义,虽然 `<<` 被赋予了新的含义,但它的优先级、操作数个数和结合性等固有的运算符属性不会改变。另外,因大于号的优先级高于条件运算符的优先级,所以,当执行 `x > y ? x : y` 时,先计算 `x > y`,再得到 `x` 或 `y`,为了使计算次序更明确,可以把 `x > y` 用圆括号括起来,即书写为 `(x > y) ? x : y`。

## 2.5 函 数

C++ 函数包括系统函数和用户函数两种,系统函数由系统定义并由相应的头文件提供函数原型,用户函数由用户在程序中定义,或者在 `#include` 命令所引用的程序文件或头文件中定义,用户函数的原型可以保存在用户建立的头文件中,也可以在程序文件的开始使用函数原型语句给出。

这一节主要介绍系统函数中的一些常用的数学函数和进行函数调用的一般格式。表 2-6 给出了每个函数的名称、原型、对应的数学表示及功能。

表 2-6 常用数学函数

函数名称	原型	数学表示	功能
整数绝对值函数	<code>int abs(int i)</code>	$ i $	返回参数 $i$ 的绝对值
实数绝对值函数	<code>double fabs(double x)</code>	$ x $	返回实数 $x$ 的绝对值
正弦函数	<code>double sin(double x)</code>	$\sin x$ ( $x$ 为弧度)	返回弧度为 $x$ 的正弦值
余弦函数	<code>double cos(double x)</code>	$\cos x$ ( $x$ 为弧度)	返回弧度为 $x$ 的余弦值
正切函数	<code>double tan(double x)</code>	$\tan x$ ( $x$ 为弧度)	返回弧度为 $x$ 的正切值
平方根函数	<code>double sqrt(double x)</code>	$\sqrt{x}$ $x > 0$	返回 $x$ 的算术平方根
指数函数	<code>double exp(double x)</code>	$e^x$ ( $e = 2.718282$ )	返回 $e^x$ 的值
幂函数	<code>double pow(double x, double y)</code>	$x^y$	返回 $x^y$ 的值
自然对数函数	<code>double log(double x)</code>	$\ln x$ ( $x > 0$ )	返回以 $e$ 为底 $x$ 的对数
对数函数	<code>double log10(double x)</code>	$\log_{10} x$ ( $x > 0$ )	返回以 10 为底 $x$ 的对数
向上取整函数	<code>double ceil(double x)</code>	$\lceil x \rceil$	返回不小于 $x$ 的最小整数
向下取整函数	<code>double floor(double x)</code>	$\lfloor x \rfloor$	返回不大于 $x$ 的最大整数
随机函数	<code>int rand(void)</code>		返回 0~32767 之间的整数
改变随机数序列	<code>void srand(unsigned s)</code>		生成与 $s$ 对应的随机数序列
终止程序运行	<code>void exit(int status)</code>		通常参数为 0 表示正常结束,非 0 表示不正常结束

在表 2-6 列出的 15 个函数中,前 12 个为数学函数,它们的函数原型包含在系统建立的 `math.h` 头文件中,后 3 个为常用的一般函数,它们的函数原型包含在系统建立的 `stdlib.h` 头

文件中。

在程序中任何位置调用一个系统函数或用户函数时,其调用格式应与它的函数原型相一致,即为:

<函数名>(<实参表达式表>)

其中所使用的圆括号为函数调用运算符,<实参表达式表>为0个(即没有)、一个或多个用逗号分隔的实参表达式,实参表达式的个数与函数原型中参数表所含的参数的个数相同。如调用 abs 函数时,实参表达式表中应当有并且只有一个参数(暂不考虑默认参数的情况);调用 pow 函数时,实参表达式表中应该包含两个表达式;调用 rand 函数时,实参表达式表应为空。

一个函数调用可以单独作为一个表达式,也可以作为表达式中的一个数据项存在,如同在表达式中使用一个常量或一个变量的情况一样。如:

```
(1) exit(1);           // 作为单独表达式使用
(2) k = abs(n1);        // 用做赋值号右边的表达式
(3) cout << sqrt(x) + 1 << endl; // 作为输出数据项中的一个数据
(4) y = 3 * exp(x/2 - 1) + a; // 作为表达式中的一个数据
(5) return pow(3,4);    // 作为返回数据
```

由于一个函数调用也是一个表达式,而有的函数是 void 类型的,调用它不返回任何值,所以这样的表达式是无值的,除此之外,表达式都是有值的。如函数调用表达式 exit(1)和 srand(10)都是无值表达式。无值表达式并不是无用的,通过函数调用虽然不返回值,但能够实现一定的操作功能。

一个函数调用中的每个实参表达式可以是任何形式的表达式,如可以是一个常量、一个变量、一个函数调用、或者一个带运算的一般表达式。如:

```
(1) abs(-24)           // 实参是一个常数
(2) abs(x)              // 实参是一个变量
(3) abs(3 * x + 4)      // 实参是一个带运算的表达式
(4) sqrt(fabs(y))       // sqrt 函数调用的实参是一个函数调用
(5) pow(x + 1, 5)        // 一个为一般表达式,另一个为常数
(6) sin(x * 3.14159/180) // 实参为一般表达式
(7) log(fabs(n1) + 7 * sqrt(n2) - 1) // 实参为带函数调用的表达式
```

当在程序运行中执行到一个函数调用时,首先依次计算出实参表中每个表达式的值,接着把每个值对应传送给函数定义参数表中的每个参数变量中,此时若实参值与参数的类型不同,则该值被自动转换成参数变量的类型后再传送,然后转去执行函数定义的定义体。当执行到定义体中的 return 语句后就结束该函数的调用过程,返回该语句中表达式(若存在的话)的值,或者当执行到定义体最后的右花括号时,也同样结束调用过程,但不会返回任何值。一个函数被调用后通常得到一个值,然后再利用这个值进行其他运算。

在一个带有函数调用的表达式中,函数调用(又称函数运算)具有最高的运算优先级,只有它被求值后才能进行以它作为一个数据项的其他运算。如计算:

```
y = a * sqrt(abs(b) - 1) + c;
```

其运算次序为:

$\text{abs}() \rightarrow - \rightarrow \text{sqrt}() \rightarrow * \rightarrow + \rightarrow =$

假定 a, b 和 c 分别为 10, 50 和 12, 则各步运算得到的值为:

$\text{abs}(50) \xrightarrow{50} - \xrightarrow{49} \text{sqrt}(49) \xrightarrow{7} * \xrightarrow{70} + \xrightarrow{82} = \xrightarrow{82} y$

最后使 y 的值为 82。

当知道了函数的原型和该函数的功能后, 用户就能够正确地调用它, 实现所具有的功能, 此时不必关心该函数的具体实现, 即函数体的具体内容。

用户在编写一个程序文件时, 要把编程需要使用的函数的原型放在程序开始和所有函数定义之前, 通过用 #include 命令中包含的头文件或单独使用的函数原型语句声明出来, 这样, 在编写每个函数定义时就能够调用这些函数。若与函数原型对应的函数定义不存在于当前程序文件中, 则它必须存在于同一程序的其他程序文件中或系统函数库中。在程序的连接阶段, 系统将使每个函数调用同相应的函数定义代码(即函数的具体实现代码)发生连接, 从而生成整个程序的可执行代码。

假定  $a = 10, b = -20, x = 3.6, y = -4.3$ , 下面给出调用表 2-6 中一些函数的返回结果。

- |  |  |
|--|--|
| (1) $\text{abs}(a) = 10$                           | // 求整数 a 的绝对值                                    |
| (2) $\text{abs}(b) = 20$                           | // 求整数 b 的绝对值                                    |
| (3) $\text{fabs}(y) = 4.3$                         | // 求实数 y 的绝对值                                    |
| (4) $\text{fabs}(a * 4 - b) = 60$                  | // 将表达式的整数值 60 自动转换为<br>// 对应的实数 60.0 后再传送该函数的参数 |
| (5) $\sin(30 * 3.14159/180) = 0.5$                 | // 求角度为 $30^\circ$ 的正弦值                          |
| (6) $\cos(30 * 3.14159/180) = 0.866026$            | // 求角度为 $30^\circ$ 的余弦值                          |
| (7) $\tan(30 * 3.14159/180) = 0.57735$             | // 求角度为 $30^\circ$ 的正切值                          |
| (8) $\text{sqrt}(36) = 6$                          | // 求 36 的平方根                                     |
| (9) $\text{sqrt}(3 * x + 1) = 3.43511$             | // 求表达式的值 11.8 的平方根                              |
| (10) $\exp(x - 2) = 4.95303$                       | // 求函数 $e^{1.6}$ 的值                              |
| (11) $\text{pow}(5, 4) = 625$                      | // 求 $5^4$ 的值                                    |
| (12) $\text{pow}(a - x, 2.5) = 103.622$            | // 求 $6.4^{2.5}$ 的值                              |
| (13) $\log(a) = 2.30259$                           | // 求函数 $\ln 10$ 的值                               |
| (14) $\log_{10}(1e - 5) = -5$                      | // 求 $\log_{10} 10^{-5}$ 的值                      |
| (15) $\log_{10}(10 * x + 50) = 1.9345$             | // 求 $\log_{10} 86$ 的值                           |
| (16) $\text{ceil}(x) = 4$                          | // 对 x 向上取整                                      |
| (17) $\text{ceil}(y) = -4$                         | // 对 y 向上取整                                      |
| (18) $\text{floor}(x) = 3$                         | // 对 x 向下取整                                      |
| (19) $\text{floor}(y) = -5$                        | // 对 y 向下取整                                      |
| (20) $\text{abs}(\text{int}(\text{floor}(y))) = 5$ | // 函数嵌套调用的情况                                     |

利用 ceil 函数或 floor 函数能够按要求取一个数的若干位小数。如, 假定  $x = 3.74265$ , 则:

- (1)  $\text{floor}(x * 100)/100 = 3.74$  // 保留 x 的两位小数, 当然 x 的值不变



- (2)  $\text{floor}(x * 10000)/10000 = 3.7426$  // 保留  $x$  的四位小数  
 (3)  $\text{floor}(x * 1000 + 0.5)/1000 = 3.743$  // 保留三位小数, 第四位四舍五入  
 (4)  $\text{floor}(x + 0.5) = 4$  // 对  $x$  取整, 小数点后一位四舍五入  
 (5)  $\text{floor}(x * 10 + 0.5) = 3.7$  // 保留一位小数, 第二位四舍五入

利用随机函数能够产生出任何指定区间内的随机数。例如:

- (1)  $\text{rand}() \% 100$  // 返回 0~99 区间内的一个随机整数  
 (2)  $10 + \text{rand}() \% 90$  // 得到 [10, 99] 区间内的一个随机整数  
 (3)  $a + \text{rand}() \% b$  // 得到 [a, a + b - 1] 区间内的一个随机整数  
 (4)  $\text{rand}() \% 100 / 100.0$  // 得到 0.00~0.99 区间内的两位随机小数  
 (5)  $\text{rand}() \% 90 / 10.0 + 1$  // 得到 1.0~9.9 区间内的含有一位整数和小数的实数

学习了 C++ 中的运算符和函数调用之后, 应能够把一个数学算式表示成正确的 C++ 算术表达式。如:

- | 数学算式                                      | C++ 算术表达式   |
|---|---|
| (1) $\frac{a+b}{a-b}$                     | $(a+b)/(a-b)$   |
| (2) $3e^x \cos(x + \pi/3)$                | $3 * \exp(x) * \cos(x + 3.14159/3)$                           |
| (3) $\frac{\sqrt{x^2 + y^2}}{2ab}$        | $\text{sqrt}(x * x + y * y) / (2 * a * b)$                    |
| (4) $-ax^{i1}(1 - x^{i2})$                | $-a * \text{pow}(x, i1) * (1 - \text{pow}(x, i2))$            |
| (5) $\frac{1}{3} \ln 14(x_1^3 + x_2 - 1)$ | $1/3.0 * \log(\text{fabs}(4 * (\text{pow}(x1, 3) + x2 - 1)))$ |

将数学算式表示成 C++ 算术表达式要注意以下几点:

- (1) 在数学算式中省略乘号的地方, 在对应的算术表达式中要加上乘号 \*;
- (2) 在数学算式的分子中若出现加、减运算, 或者在分母中若出现加、减、乘和除运算, 则表示成算术表达式时应把分子和分母分别用圆括号括起来;
- (3) 对于在三角函数中使用的角度, 应转换为弧度, 对于使用的常量符号  $\pi$  应写成常数 3.141 59;
- (4) 当算式中带有两个整数相除时, 则转换后要使其一写成实数形式, 否则将丢失商的小数部分, 使运算结果不正确;
- (5) 在编程时可以把数学算式中使用的变量直接定义为 C++ 变量, 也可以使用其他标识符作为相应变量的变量名。如把数学算式(1)中的  $a$  和  $b$  分别用 C++ 变量  $x$  和  $y$  来表示, 则对应的算术表达式为  $(x+y)/(x-y)$ 。

下面给出两个使用系统函数的程序例子。

程序 1:

```
#include <iostream.h>
#include <math.h>
void main()
{
    int a = 15;
    double x = 4.26;
    cout << "a, x = " << a << ', ' << x << endl;
```

```

    cout << "abs(a),abs(x) =" << abs(a) << ', ' << abs(x) << endl;
    cout << "fabs(a),fabs(x) =" << fabs(a) << ', ' << fabs(x) << endl;
    cout << "floor(a),floor(x) =" << floor(a) << ', ' << floor(x) << endl;
    cout << "sqrt(a),sqrt(x) =" << sqrt(a) << ', ' << sqrt(x) << endl;
    cout << "pow(a,2),pow(x,3) =" << pow(a,2) << ', ' << pow(x,3) << endl;
}

```

该程序运行结果为:

```

a,x=15,4.26
abs(a),abs(x) = 15,4
fabs(a),fabs(x) = 15,4.26
floor(a),floor(x) = 15,4
sqrt(a),sqrt(x) = 3.87298,2.06398
pow(a,2),pow(x,3) = 225,77.3088

```

程序 2:

```

#include<iostream.h>
#include<stdlib.h>
#include<time.h>
void main()
{
    int x,y,z;
    srand(time(0)); // time 函数原型包含在系统头文件 time.h 中,
    // 它返回从 1970 年 1 月 1 日零时算起至当前时间为止的秒数,
    // 由于当前时间是时刻变化的,所以可使每次运行程序时调用
    // srand 函数的实参值均不同,从而使系统生成每次均不同
    // 的随机数序列,使下面的 x 和 y 每次均得到不同的整数。
    x = rand() % 100;
    y = rand() % 100;
    cout << x << "+" << y << "=";
    cin >> z;
    if(x+y == z) cout << "回答正确!" << endl;
    else cout << "回答错误!" << endl;
}

```

该程序的两次运行结果为:

```

35 + 73 = 108          // 108 及后面的回车是用户的输入
回答正确!
98 + 37 = 125          // 125 及后面的回车是用户的输入
回答错误!

```

## 习题二

### (一) 简答题

1. C++ 数据分为哪四种大的类型?
2. 在大的整数类型中又分为哪四种类型?

3. 整数类型分为哪三种类型,每种类型的长度各是多少?
4. 实数类型分为哪三种类型,它们的类型长度各是多少?
5. 单精度和双精度分别表示多少位有效数字?
6. 每一种C++ 具体类型的关键字是什么?
7. 哪四种保留字能够用来修饰 int 保留字成为类型关键字? 哪两种保留字能够用来修饰 char 保留字成为类型关键字?
8. 整型常数具有哪三种不同进制的表示? 如何区分它们?
9. 一个整数后缀 L 或 U 分别具有什么含义?
10. 转义字符序列中的首字符是什么字符?
11. 实数具有哪两种表示方法? 一个浮点数具有哪些成分? 什么叫规格化浮点数?
12. 定义符号常量和变量的语句有什么异同?
13. #define 命令的作用是什么?
14. 一个C++ 运算符具有哪三种属性?
15. 左值和非左值(又称右值)有什么区别?
16. 在程序文件中使用系统数学函数或随机函数时,各需要通过 #include 命令引入哪个头文件?
17. 一个表达式是否必须有一个值,在什么情况下它没有值?
18. 执行一个函数调用的大致过程是什么? 函数运算的优先级如何?

## (二) 填空题

1. 数据类型 int, char, bool, float, double 等的类型长度分别为\_\_\_\_、\_\_\_\_、\_\_\_\_、\_\_\_\_和\_\_\_\_。
2. 短整型、整型和长整型的最简关键字分别为\_\_\_\_、\_\_\_\_和\_\_\_\_。
3. 常数 -4.205, 1200 和 6.7E-9 分别具有\_\_\_\_、\_\_\_\_和\_\_\_\_位有效数字。
4. 数值常量 46、0173 和 0x62 对应的十进制值分别为\_\_\_\_、\_\_\_\_和\_\_\_\_。
5. 字符常量 'k', '\n', '\\', '/052' 和 '\xA2' 对应的数值分别为\_\_\_\_、\_\_\_\_、\_\_\_\_、\_\_\_\_和\_\_\_\_。
6. 字符串 "It's \40an \40apple. \n" 中包含有\_\_\_\_个字符。
7. 枚举类型中的每个枚举值都是一个\_\_\_\_, 它的值是一个\_\_\_\_, 值的类型为\_\_\_\_。
8. 实数 340.0 和 0.027E8 对应的规格化浮点数分别为\_\_\_\_和\_\_\_\_。
9. 常数 100, -25, 3.62, 1E5 和 -4.73f 的数据类型分别为\_\_\_\_、\_\_\_\_、\_\_\_\_、\_\_\_\_和\_\_\_\_。
10. 若 x 的值为 10, 则 x+=5 的值和运算后的\_\_\_\_的值相同, 等于\_\_\_\_。
11. 若 x=5, y=10, 则计算 y\*=++x 表达式后, x 和 y 的值分别为\_\_\_\_和\_\_\_\_。
12. 若 x=25, 则计算 y=x++ 表达式后, x 和 y 的值分别为\_\_\_\_和\_\_\_\_。

13. 假定  $x$  和  $ch$  分别为 `int` 型和 `char` 型, 则 `sizeof(x)` 和 `sizeof(ch)` 的值分别为\_\_\_\_\_和\_\_\_\_\_。
14. 假定  $x=64$ ,  $y=88$ , 则  $x << 2$  和  $y >> 2$  的值分别为\_\_\_\_\_和\_\_\_\_\_。
15. 假定  $x$  是一个逻辑量, 则  $x \&\& \text{true}$  的值与\_\_\_\_\_的值相同,  $x || \text{false}$  的值也与\_\_\_\_\_的值相同。
16. 假定  $x$  是一个逻辑量, 则  $x \&\& !x$  和  $x || !x$  的值分别为\_\_\_\_\_和\_\_\_\_\_。
17. 假定  $x=10$ , 则表达式  $x \leq 10 ? 20 : 30$  的值为\_\_\_\_\_。
18. 假定  $x=10.5$ ,  $y=-4.6$ , 则表达式 `floor(fabs(x+y))` 的值为\_\_\_\_\_。
19. 表达式 `sqrt(81)` 和 `pow(6,3)` 的值分别为\_\_\_\_\_和\_\_\_\_\_。
20. 随机函数 `rand()` % 20 的值在\_\_\_\_\_至\_\_\_\_\_区间内。
21. 数学算式  $(1+x) \sin 48^\circ$  和  $ax^b e^{x+1}$  对应的算术表达式分别为\_\_\_\_\_和\_\_\_\_\_。

(三) 假定  $a$  为 `int` 型,  $x$  为 `double` 型, 试指出下列各表达式值的类型

- |                                   |                                |                            |
|-----------------------------------|--------------------------------|----------------------------|
| 1. 327                            | 2. -1.5e6                      | 3. 42.0f                   |
| 4. 48L                            | 5. 0372                        | 6. 0xabce                  |
| 7. 1/3                            | 8. 1%3                         | 9. 1.0%3                   |
| 10. $a * 3 / x$                   | 11. $a * (a + x)$              | 12. $1 + a * a$            |
| 13. $a += 3$                      | 14. $x * = 2$                  | 15. 'x'                    |
| 16. 'x' + 20                      | 17. true                       | 18. ++a                    |
| 19. short(a)                      | 20. bool(x)                    | 21. (unsigned int) (0256)  |
| 22. $x = a ++$                    | 23. sizeof(bool)               | 24. char(a + 10)           |
| 25. abs(x)                        | 26. fabs(a)                    | 27. floor(x)               |
| 28. ceil(a)                       | 29. $\exp(4) + 2$              | 30. rand()                 |
| 31. <code>rand()</code> / 100     | 32. <code>floor(log(x))</code> | 33. <code>pow(a, 4)</code> |
| 34. $a > 10 ? x : \text{sqrt}(x)$ | 35. $a > 100$                  | 36. $x! = 10$              |
| 37. $a \&\& x$                    | 38. $a < 1    a > 10$          | 39. !x                     |
| 40. ! (a == 0)                    |                                |                            |

(四) 已知  $a=20$ ,  $x=4.7$ ,  $r='a'$ , 试求出下列每个表达式的值(各表达式互不影响)

- |                                      |                                      |
|--------------------------------------|--------------------------------------|
| 1. $a ++$                            | 2. ++r                               |
| 3. $a \% 5$                          | 4. $a / 5$                           |
| 5. $r \% 18$                         | 6. $1 + a / 3 \% 4$                  |
| 7. $a / 8.0$                         | 8. $r += 4$                          |
| 9. $x = 2 * (x + 5) - 1$             | 10. char(a + 30)                     |
| 11. $10 * \text{sizeof}(\text{int})$ | 12. $(a << 3) + 5$                   |
| 13. $a > 0$                          | 14. $a == 1$                         |
| 15. $a ++ \&\& r! = 'a'$             | 16. ++a && r == 'a'                  |
| 17. $a    x \leq 10$                 | 18. $a == r    a == \text{floor}(x)$ |

19.  $a = (r > 'A' ? a + 10 : a - 10)$

21.  $x = (a ++ , r ++)$

23.  $x = \text{pow}(3, 5)$

25.  $a = \text{abs}(a) \% 8 - 10$

27.  $\text{ceil}(x - 0.5)$

20.  $a = (r < 'A' ? a + 10 : a - 10)$

22.  $x = (a ++ , r ++ , 50)$

24.  $x = \text{fabs}(-26.5) + 4$

26.  $\text{floor}(x + 0.5)$

28.  $\text{ceil}((x - 0.5) - 0.5)$

(五) 给出下列每个逻辑表达式的相反式

1.  $x$

2.  $x == 0$

3.  $a! = \text{true}$

4.  $x \geq 10$

5.  $x! = \text{key} \ \&\& \ \text{flag}$

6.  $p! = \text{NULL} \ \&\& \ \text{flag} == \text{true}$

7.  $x > 0 \ \&\& \ x < 10$

8.  $x == \text{key} \ || \ \text{true}$

9.  $x \geq a \ || \ \text{false}$

10.  $\text{ch} == 'd' \ || \ \text{ch} == 'D'$

11.  $!p \ \&\& \ p -> \text{data}! = x$

12.  $i < n \ \&\& \ a[i]! = \text{key}$

13.  $\text{ch} == '(' \ || \ \text{ch} == '[' \ || \ \text{ch} == '|'$

14.  $x \ || \ i < n \ \&\& \ i! = 0$

(六) 把下列数学算式或不等式表示成C++表达式

1.  $2x \left( 1 + \frac{x^2}{3} \right)$

2.  $\frac{1+e^x}{1-e^x}$

3.  $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$

4.  $\frac{1}{3^x \ln(2x+k)}$

5.  $\frac{\sin^3(x + \pi/4)}{3 + \cos^3(x - \pi/4)}$

6.  $\frac{1}{7} (1 + e^{x+1})^n$

7.  $0 \leq x \leq 20$

8.  $ax - by \neq c$

9.  $4x + 7y - 2 = 3ab$

10.  $\left| \frac{2x^2 + 1}{3x + 2} \right| \leq 5$  同时  $3x + 2 \neq 0$

11.  $\text{age} \geq 55$  或者  $\text{pay} \geq 820$

12. place = "江苏" 同时 sex = "女"
13. 'a' ≤ ch ≤ 'z' 或者 'A' ≤ ch ≤ 'Z'
14. s[0] = '0' 同时 (s[1] = 'x' 或者 s[1] = 'X')

(七) 写出下列每个程序运行后的输出结果并自行上机验证

1. 

```
#include<iostream.h>
enum race{Black, White, Yellow};
void main()
{
    race ra;
    ra = Black;cout << ra << ',';
    ra = White;cout << ra << ',';
    ra = Yellow;cout << ra << ',';
    cout << Black+White+Yellow << endl;
}
```
2. 

```
#include<iostream.h>
#define PI 3.14159
const int R=10;
void main()
{
    double p,s;
    p=2*R*PI;
    s=R*R*PI;
    cout <<"p="<<p << endl;
    cout <<"s="<<s << endl;
}
```
3. 

```
#include<iostream.h>
void main()
{
    int i=0,j=1,k;
    k=i+=j;
    cout <<i <<' '<<j <<' '<<k << endl;
    k=(i++)*(++j);
    cout <<i <<' '<<j <<' '<<k << endl;
    k*=i++*j--;
    cout <<i <<' '<<j <<' '<<k << endl;
}
```
4. 

```
#include<iostream.h>
void main()
{
    cout << sizeof(bool) <<' '<< sizeof(char) <<' ';
    cout << sizeof(short) <<' '<< sizeof(int) <<' ';
    cout << sizeof(long) <<' '<< sizeof(float) <<' ';
    cout << sizeof(double) <<' '<< sizeof(long double) <<' ';
    cout << sizeof(int*) <<' '<< sizeof(double*) << endl;
}
```

5. 

```
#include <iostream.h>
const int w=30;
void main()
{
    int x,y;
    x=w << 2; y=w >> 2;
    cout << w << ' ' << x << ' ' << y << endl;
    x=w&10; y=w | 10;
    cout << w << ' ' << x << ' ' << y << endl;
    x=w/10; y= ~x & w;
    cout << w << ' ' << x << ' ' << y << endl;
}
```
6. 

```
#include <iostream.h>
void main()
{
    int x=10, y=-1;
    cout << ((x>y) && (y<0)) << ' ';
    cout << ((x>y) || (y<0)) << ' ';
    cout << ((x<=y) && (y>=0)) << ' ';
    cout << ((x<=y) && (y>=0)) << ' ';
    cout << ((x==y) && y) << ' ';
    cout << ((x==y) || y) << endl;
}
```
7. 

```
#include <iostream.h>
void main()
{
    int x=5, y=10, k;
    (k=x) += y; cout << x << ' ' << y << ' ' << k << endl;
    k=x+=y; cout << x << ' ' << y << ' ' << k << endl;
    k=x*=y; cout << x << ' ' << y << ' ' << k << endl;
    k=(x++, y++); cout << x << ' ' << y << ' ' << k << endl;
    k=x++, y++;
    cout << x << ' ' << y << ' ' << k << endl;
}
```
8. 

```
#include <iostream.h>
#include <math.h>
void main()
{
    double x=15.72;
    double y,z;
    y=ceil(x); z=floor(x);
    cout << x << ' ' << y << ' ' << z << endl;
    y=floor(x+0.5); z=floor(x*10+0.5)/10;
    cout << x << ' ' << y << ' ' << z << endl;
    y=ceil(x-0.5); z=ceil(x*10-0.5)/10;
    cout << x << ' ' << y << ' ' << z << endl;
}
```



#### (八) 编写下列程序

1. 已知一个三角形中三条边的长度分别为  $a$ ,  $b$  和  $c$ , 则利用计算公式  $\sqrt{s(s-a)(s-b)(s-c)}$  求出三角形的面积, 其中  $s = (a+b+c)/2$ , 假定  $a, b$  和  $c$  的值由键盘输入, 并确保任何两边的长度大于等于第三条边。

2. 假定一所大学 2001 年招生人数为 3000 人, 若以后每年平均比上一年计划扩招 10%, 则 2006 年将计划招生多少人?

3. 已知有 4 个整数为  $a, b, c, d$ , 试计算出它们的算术平均值和几何平均值。

4. 已知  $x = \frac{2a}{3(a+b)} \sin a$ ,  $y = \frac{2b}{3(a+b)} \cos b$ , 试根据从键盘上输入的  $a$  和  $b$  的值分别计算出  $x$  和  $y$  的值。

#### (九) 上机实验题

1. 让计算机随机出 10 道两位整数加法题供用户计算, 每道题 10 分, 计算完成后打印出得分。

```
1. #include <iostream.h>
   #include <stdlib.h>
   #include <time.h>
   const N=10;
   void main()
   {
       srand(time(0));
       int x,y,z,c=0;
       for(int i=1; i <= N; i++) {
           x=rand() % 90 + 10;
           y=rand() % 90 + 10;
           cout << x << ' + ' << y << ' = ';
           cin >> z;
           if(x+y == z) c++;
       }
       cout << "得分:" << c * 10 << endl;
   }
```

2. 打印出  $0^\circ \sim 90^\circ$  之间每隔  $5^\circ$  的正弦值和余弦值。

```
#include <iomanip.h>
#include <math.h>
const double RAD = 3.14159/180;
void main()
{
    int i=0;
    while(i <= 90) {
        cout << setw(5) << i << setw(10) << sin(i * RAD) << ' ';
        // setw(n)使后面一个数据的输出宽度为 n,
        // 它在 iomanip.h 头文件中有定义
        cout << setw(10) << cos(i * RAD) << endl;
        i += 5;
    }
```

```

    }
}

```

3. 把从键盘上输入的一个正整数按数字位的相反次序输出。

```

#include <iostream.h>
void main()
{
    int num, rem;
    cout << "输入一个整数:";
    cin >> num;
    do {
        rem = num % 10;
        num /= 10;
        cout << rem;
    } while (num > 0);
    cout << endl;
}

```

4. 打印出 a~f 中每个大小写字母所对应的 ASCII 码。

```

#include <iostream.h>
void main()
{
    char c1 = 'A', c2 = 'a';
    for (int i = 1; i <= 6; i++) {
        cout << c1 << ':' << int(c1) << ", ";
        cout << c2 << ':' << int(c2) << endl;
        c1++; c2++;
    }
}

```

5. 从键盘上输入两个整数, 由用户回答它们的和、差、乘、除和取余运算的结果, 并统计出正确答案的个数。

```

#include <iostream.h>
void main()
{
    int x, y, z, c = 0;
    cout << "输入两个整数:";
    cin >> x >> y;
    cout << x << ' + ' << y << ' = ';
    cin >> z; if (x + y == z) c++;
    cout << x << ' - ' << y << ' = ';
    cin >> z; if (x - y == z) c++;
    cout << x << ' * ' << y << ' = ';
    cin >> z; if (x * y == z) c++;
    cout << x << ' / ' << y << ' = ';
    cin >> z; if (x / y == z) c++;
    cout << x << ' % ' << y << ' = ';
    cin >> z; if (x % y == z) c++;
    cout << "答对" << c << "道题" << endl;
}

```

## 第三章 流程控制语句

流程控制语句用来控制程序的执行流程,它包括选择、循环和跳转三类语句。

选择类语句包括 if 语句和 switch 语句两种,用它们来解决实际应用中按不同情况进行不同处理的问题。如当调整职工工资时,应按不同的级别增长工资;大学生交纳学费时,应按不同的专业交纳不同的学费。

循环类语句包括 for 循环语句、while 循环语句和 do 循环语句三种,用它们来解决实际应用中需要重复处理的问题。如当统计全体职工工资总和时,就需要重复地做加法,依次把每个人的工资累加起来;当从一批数据中查找具有最大值的一个数据时,需要重复地做两个数的比较运算,每次把上一次比较得到的大者同一个新(即未比较)的数据比较,当同最后一个新的数据比较后得到的大者就是全部数据中的最大值。

跳转类语句包括 goto 语句、continue 语句、break 语句和 return 语句四种,用它们来改变顺序向下执行的正常次序,而转向隐含或显式给出的语句位置,接着从此位置起向下执行。如当从一批数据中查找一个与给定值相等的数据时,最简单的方法是从前向后使每一个数据依次同给定值进行比较,若不等则继续向下比较,若相等则表明查找成功,应终止比较过程,此时就需要使用跳转语句转移到其他地方执行。

这一章将依次介绍每一种流程控制语句的语法格式、执行过程和应用举例等内容。

### 3.1 if 语句

#### 1. 语句格式

if 语句又称条件语句,其语句格式为:

```
if (<表达式>) <语句 1> [else <语句 2>]
```

if 语句是一种结构性语句,因为它又包含有语句,即 <语句 1> 和可选择的 <语句 2>,这两条语句称为 if 语句的子句。

在 if 语句格式中,其后的保留字 else 和 <语句 2> 是任选项,带与不带都是允许的。

if 语句中的每个子句可以是任何可执行语句或空语句,可执行语句包括表达式语句、复合语句、以及任一种流程控制语句等。

#### 2. 语句执行过程

if 语句的执行过程为:

(1) 求 <表达式> 的值,若它的值非 0,则表明 <表达式> (又称为条件)为真或成立,否则认为条件为假或不成立;

(2) 当条件为真则执行 < 语句 1 >, 为假则执行 < 语句 2 >, 但若 else 部分被省略, 则不会执行任何操作。

执行 if 语句的过程可用图 3-1 描述, 其中菱形框表示判断, 矩形框表示处理, 带箭头的连线表示执行走向。图 3-1(a) 和 (b) 分别表示省略和带有 else 部分的具体执行流程。

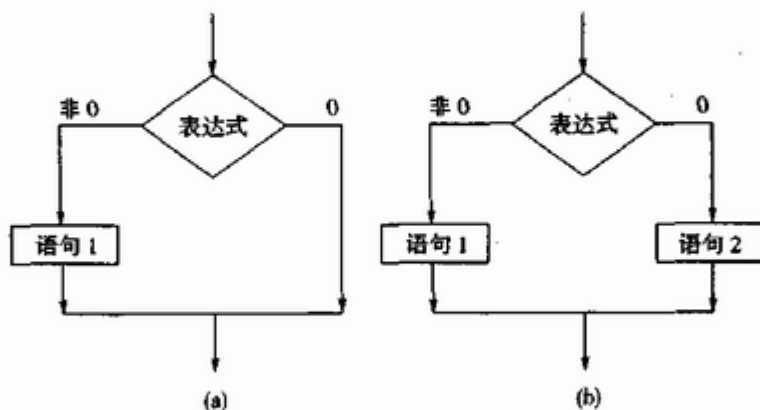


图 3-1 if 语句执行流程示意图

### 3. 语句格式举例

- (1) `if(x != -1) c++;`
- (2) `if(x <= a) s1 += x; else s2 += x;`
- (3) `if(fabs(x) <= 1) y = 1 + exp(x);`  
`else y = 1 + 2 * x;`
- (4) `if(grade >= 60 && grade <= 100) cout << "pass" << endl;`
- (5) `if(grade < 0 || grade > 100) cout << "Score error!" << endl;`
- (6) `if(p && a > b) cout << "a > b" << endl;`  
`else cout << "a <= b" << endl;`
- (7) `if(x * x + y * y == z * z) {c++; w = x + y + z;}`
- (8) `if(x) {y = 3 * x - 1; z = sqrt(fabs(x)) + 2;} else {y = 6; z = y * pow(y, 4) - 3;}`

在以上列举的语句中, 作为判断条件的表达式有的为单个变量, 有的为关系表达式, 有的为逻辑表达式, 作为子句的 < 语句 1 > 或 < 语句 2 > 有的为简单语句, 有的为复合语句。每条语句的执行过程一目了然, 如执行第一条语句时, 若  $x$  不等于  $-1$  成立, 则执行  $c++$  操作, 否则不执行任何操作; 执行第二条语句时, 若  $x$  小于等于  $a$  成立, 则执行  $s1 += x$  操作, 否则执行  $s2 += x$  操作; 执行第八条语句时, 若  $x$  不为 0, 则执行格式中 < 语句 1 > 所对应的复合语句, 否则执行 < 语句 2 > 所对应的复合语句。

### 4. 语句嵌套

if 语句中的任何一个子句可以为任何可执行语句, 当然仍可以是一条 if 语句, 此种情况称为 if 语句的嵌套。当出现 if 语句嵌套时, 不管书写格式如何, else 都将与它前面最靠近的 if 相配对, 构成一条完整的 if 语句。如:

- (1) if( <表达式 1> ) if( <表达式 2> ) <语句 1> else <语句 2>
- (2) if( <表达式 1> ) { if( <表达式 2> ) <语句 1> <语句 2> } else <语句 3>
- (3) if( <表达式 1> ) <语句 1>  
    else if( <表达式 2> ) <语句 2>  
    else <语句 3>
- (4) if( <表达式 1> ) <语句 1>  
    else if( <表达式 2> ) <语句 2>  
    else if( <表达式 3> ) <语句 3>  
    else <语句 4>

注意：在第二条语句中，else 不是同它前面复合语句中的 if 相配对，而是与处于同一层次的最前面的 if 相配对。

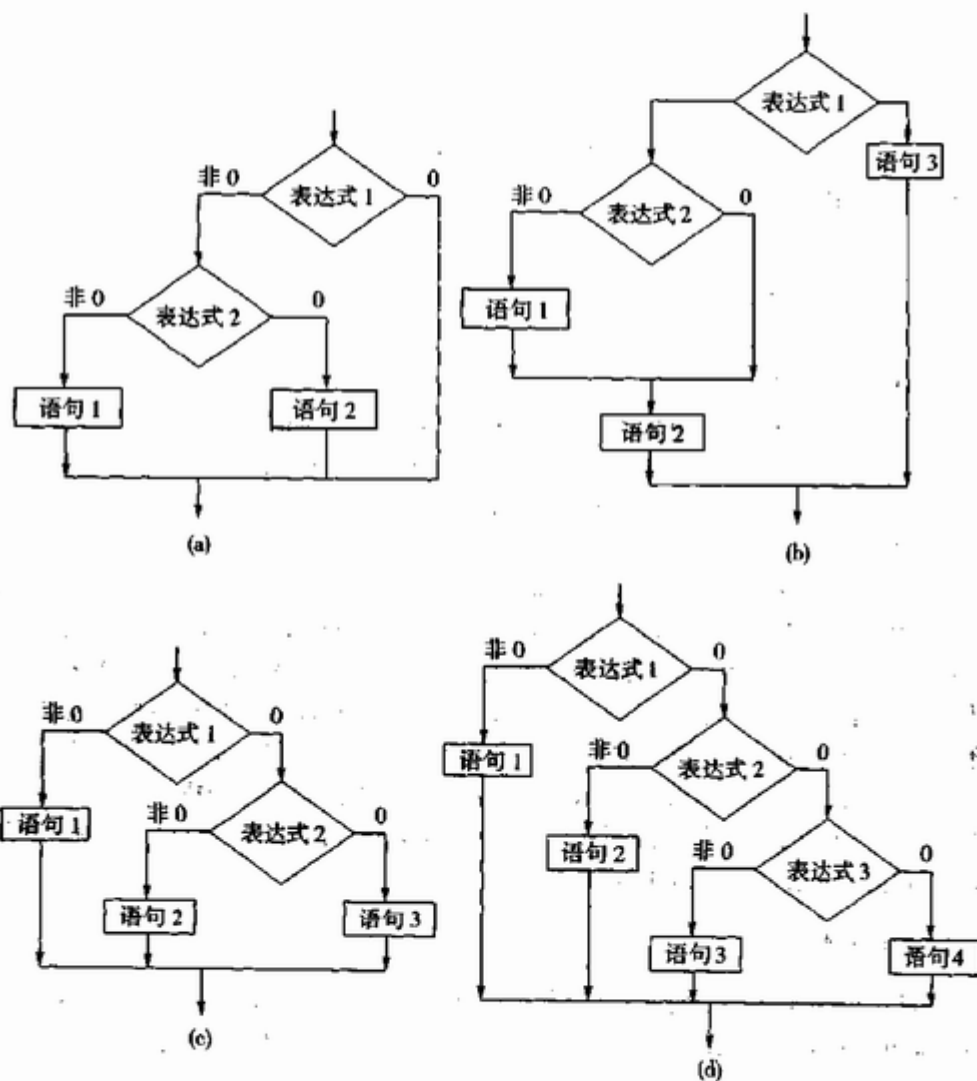


图 3-2 嵌套 if 语句的执行流程

以上每条语句的执行过程如图 3-2(a) ~ (d)所示。

## 5. 程序举例

```
(1) #include <iostream.h>
void main()
{
    int x,y;
    cout << "从键盘输入一个整数: ";
    cin >> x;
    if(x<0) y=1+2*x*x;
    else y=7*x-4;
    cout << "x=" << x << ", " << "y=" << y << endl;
}
```

该程序的功能是:根据从键盘上输入的  $x$  的值计算并输出  $y$  的值, $y$  的计算公式为:

$$y = \begin{cases} 1+2x^2 & (x<0) \\ 7x-4 & (x\geq 0) \end{cases}$$

```
(2) #include <iomanip.h>
#include <math.h>
void main()
{
    double x,y;
    cin >> x;
    if(x<0) y=fabs(x);
    else if(x<10) y=exp(x)*sin(x);
    else if(x<20) y=pow(x,3);
    else y=(3+2*x)*log(x);
    cout << setw(10) << x << setw(10) << y << endl;
    //分别使 x 和 y 的输出宽度为 10,即占有 10 个字符位置
}
```

该程序的功能是:根据  $x$  的值计算出分段函数  $y$  的值, $y$  的计算公式为:

$$y = \begin{cases} |x| & (x<0) \\ e^x \sin x & (0\leq x<10) \\ x^3 & (10\leq x<20) \\ (3+2x)\ln x & (x\geq 20) \end{cases}$$

```
(3) #include <iostream.h>
void main()
{
    int a,b,c,temp;
    cout << "输入三个整数: ";
    cin >> a >> b >> c;
    if(a<b) {temp=a; a=b; b=temp;}
    if(a<c) {temp=a; a=c; c=temp;}
    if(b<c) {temp=b; b=c; c=temp;}
    cout << a << ' ' << b << ' ' << c << endl;
}
```

该程序的功能是把从键盘上输入的按任意次序排列的三个整数转变为按从大到小的次序排列(即  $a \geq b \geq c$ )并输出出来。

**注意:** 对于该程序中每条 if 语句中的复合语句,其作用是交换两个变量的值,它首先把第一个变量的值暂存到 temp 变量中,接着把第二个变量的值赋给第一个变量,最后把 temp 变量的值,即第一个变量的原值赋给第二个变量中。若不通过中间变量 temp,而是直接把第一个变量的值赋给第二个变量,再把第二个变量的值赋给第一个变量,则不能够达到交换两个变量值的目的,请读者思考!

## 3.2 switch 语句

### 1. 语句格式

switch 语句又称情况语句或开关语句,它也是一种结构性语句,其语句格式为:

switch (<表达式>) <语句>

该语句中所包含的 <语句> 通常是一条复合语句,并在内部的一些语句前加有特殊的语句标号“case <常量表达式>:”或“default:”,因此,switch 语句的实际使用格式为:

```
switch(<表达式>){
    case <常量表达式1>: <语句1-1>
                        <语句1-2>
                        ⋮
    case <常量表达式2>: <语句2-1>
                        <语句2-2>
                        ⋮
                        ⋮
    [default: <语句n-1>
              <语句n-2>
              ⋮
    ]
}
```

该语句中可以使用一次或多次 case 标号,但只能使用一次 default 标号,或者省略掉整个 default 部分。另外,多个 case 标号也允许使用在同一条语句的前面。

**注意:** 语句标号只起到标识语句位置的作用,对语句的执行不会产生任何影响。

### 2. 语句执行过程

switch 语句的执行过程为:

(1) 计算出 <表达式> 的值,假定为 M,若它不是整型,系统将自动舍去其小数部分,只取其整数部分作为结果值;

(2) 依次计算出每个常量表达式的值,假定它们的值依次为 M1, M2, ... 同样若它们的值不是整型,则自动转换为整型;



(3) 让  $M$  依次同  $M_1, M_2, \dots$  进行比较, 一旦遇到  $M$  与某个值相等, 则就从对应标号的语句开始向下执行, 若碰不到跳转语句的话, 将一直执行到右花括号为止才结束整个 `switch` 语句的执行, 若  $M$  与所有值都不同, 则当带有 `default` 部分时, 就从该标号位置起向下执行, 否则不执行任何操作。

执行 `switch` 语句的过程可用图 3-3 加以描述。

在实际使用 `switch` 语句时, 通常要求当执行完某个语句标号开始的一组语句后, 就结束整个语句的执行, 而不让它继续执行下一个语句标号后面的语句序列, 为此, 可通过使用 `break` 语句来实现。该语句只有保留字 `break`, 而没有其他任何成分。它是一条跳转语句, 在 `switch` 语句中执行到它时, 将跳转到所属的 `switch` 语句的后面位置, 系统将接着向下执行其他语句。

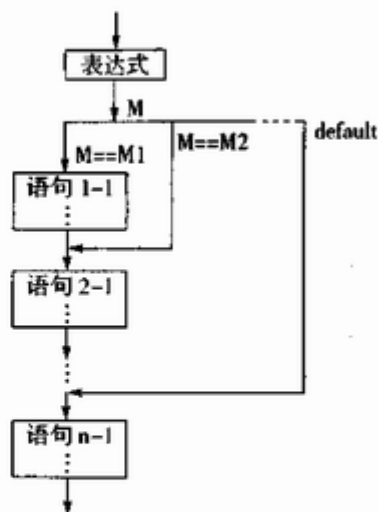


图 3-3 `switch` 语句执行流程

### 3. 语句格式举例

- (1) `switch(a) {`  
`case 1: c1++; break;`  
`case 2: c2++; break;`  
`case 3: c3++; break;`  
`case 4: c4++; break;`  
`default: c++; break;`  
`}`
- (2) `switch(cr) {`  
`case red: cout << "red" << endl; break;`  
`case yellow: cout << "yellow" << endl; break;`  
`case blue: cout << "blue" << endl;`  
`}`
- (3) `switch(ch) {`  
`case 'a':`  
`case 'A': d1 = (x+y)/2;`  
`d2 = x*y - 2;`  
`break;`  
`case 'b':`  
`case 'B': d1 = (a+b)/2;`  
`d2 = a*b - 2;`  
`break;`  
`default: cout << "Input error!" << endl;`  
`exit(1);`  
`}`

第一条语句执行时, 将按照  $a$  的取值使相应的变量增 1, 具体地说, 当  $a$  取 1 时  $c_1$  增 1,  $a$

取 2 时 c2 增 1, a 取 3 时 c3 增 1, 取 4 时 c4 增 1, a 取其他任何值时则使变量 c 增 1, 每执行增 1 操作后, 都接着执行一条 break 语句, 使执行流程转出整个 switch 语句, 否则将会顺序执行后面的增 1 语句。

执行第二条语句时, 将按照具有枚举类型 color 的变量 cr 的值决定输出哪一个常量标识符, 当 cr 取值为 red(即 0)时输出 red 标识符, 取值为 yellow(即 1)时输出 yellow 标识符, 取值为 blue(即 2)时输出 blue 标识符。输出最后一个常量标识符虽然没有使用 break 语句转出去, 但由于它后面就是语句结束标志, 右花括号, 所以也会自然地结束该语句。

当执行第三条语句时, 若 ch 值为小写字母 a 或大写字母 A, 则执行 3~5 行的语句, 若 ch 值为小写字母 b 或大写字母 B, 则执行 7~9 行的语句, 若 ch 不是上述取值, 则执行 10~11 行后结束整个程序的运行。

在 switch 语句所含的复合语句中, 可以包含任何语句, 当然仍可以是 switch 语句, 所以 switch 语句也允许出现嵌套的情况。

#### 4. 程序举例

```
(1) #include <iostream.h>
void main()
{
    int weekday;
    cout << "今天星期几(0-6)? ";
    cin >> weekday;
    switch(weekday) {
        case 0: cout << "sunday" << endl; break;
        case 1: cout << "Monday" << endl; break;
        case 2: cout << "Tuesday" << endl; break;
        case 3: cout << "Wednesday" << endl; break;
        case 4: cout << "Thursday" << endl; break;
        case 5: cout << "Friday" << endl; break;
        case 6: cout << "Saturday" << endl; break;
        default: cout << "Input error!" << endl;
    }
}
```

该程序的功能是: 根据从键盘上输入的代表星期几的数字, 对应输出它的英文名称。

```
(2) #include <iostream.h>
#include <stdlib.h>
void main()
{
    float score;
    cout << "输入一个人的成绩: ";
    cin >> score;
    if(score < 0 || score > 100) {
        cout << "输入数据有误!" << endl;
        exit(1);
    }
    switch(int(score)/10) {
        case 9:
```

```

        case 10: cout << score << ":优" << endl; break;
        case 8: cout << score << ":良" << endl; break;
        case 7: cout << score << ":中" << endl; break;
        case 6: cout << score << ":及格" << endl; break;
        default: cout << score << ":不及格" << endl; break;
    }
}

```

该程序的功能是:根据从键盘上输入的一个人的成绩判断并输出它所属的等级。等级分为优、良、中、及格和不及格五个级别,对应的分数段依次为[90,100], [80,89], [70,79], [60,69]和[0,59]。

```

(3) #include <iostream.h>
    #include <stdlib.h>
    #include <time.h>
    void main()
    {
        char mark;
        int x,y,z;
        bool b=false;
        srand(time(0)); //初始化系统中的随机数序列
        x=rand() % 50 + 1;
        y=rand() % 10 + 1;
        cout << "输入一个算术运算符(+, -, *, /, %): ";
        cin >> mark;
        cout << x << mark << y << '=';
        cin >> z;
        switch(mark) {
            case '+': if(z == x+y) b=true; break;
            case '-': if(z == x-y) b=true; break;
            case '*': if(z == x*y) b=true; break;
            case '/': if(z == x/y) b=true; break;
            case '%': if(z == x%y) b=true; break;
            default: cout << "运算符输入错!" << endl;
                    exit(1);
        }
        if(b) cout << "right!" << endl;
        else cout << "error!" << endl;
    }

```

该程序的功能是:首先让计算机产生出两个随机整数  $x$  和  $y$ ,  $x$  在 1~50 以内,  $y$  在 1~10 以内;接着由用户输入一个运算符,再由用户输入对  $x$  和  $y$  的运算结果;然后判断用户的计算是否正确,若正确则置  $b$  为 true,即 1,否则保持原值 0 不变;程序最后输出相应的信息表示计算正确或错误。

### 3.3 for 语句

#### 1. 语句格式

for 语句又称 for 循环,它也是一种结构性语句,其语句格式为:

```
for(<表达式 1>; <表达式 2>; <表达式 3>) <语句>
```

其中<语句>是 for 语句的循环体,它将按条件被重复执行多次;<表达式 1>,<表达式 2>和<表达式 3>都可以被省略,但它们之间的分隔符(即分号)必须保留;另外,<表达式 1>除了可以是一个表达式外,还可以兼有对变量进行定义的功能,此变量在离开此循环后仍然可以使用。如  $i=1$  和  $\text{int } i=1$  都可以作为<表达式 1>使用,当使用  $i=1$  时, $i$  必须被定义过,当使用  $\text{int } i=1$  时, $i$  在此之前必须没有定义,此表达式同时具有定义变量  $i$  和给它赋初值这两种功能。

#### 2. 语句执行过程

for 语句的执行过程为:

- (1) 计算<表达式 1>,当然若此项被省略则无须计算;
  - (2) 计算<表达式 2>得到一个值,假定为  $M$ ,若该表达式被省略则当作数值 1 看待;
  - (3) 若  $M$  为非 0,则执行一遍循环体,否则结束整个 for 语句的执行;
  - (4) 计算<表达式 3>,当然若此项被省略则无须计算;
  - (5) 自动转向第(2)步执行。
- 执行 for 循环的过程如图 3-4 所示。

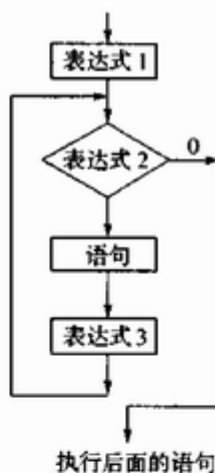


图 3-4 for 语句执行流程

#### 3. 语句格式举例

- (1) `for(i=1; i<10; i++) cout << i << ' ';`
- (2) `for(int i=1; i++ <= 1000;);`
- (3) `for(int i=0, j=0; i+j<20; i++, j+=2) x=i*i+j*j;`
- (4) `for(;;) {i++; if(i>100) break;}`
- (5) `for(i=0, y=0; i<n; i++) {`  
    `cin >> x;`  
    `y += x;`  
    `}`
- (6) `for(int k=2; k<sqrt(m); k++)`  
    `if(m%k == 0) break;`
- (7) `for(; b; a=b, b=r) r=a%b;`
- (8) `for(k=20; k!=0; k--)`

```

a = rand() % 100;
cout << a << ' ';
if(a%2) c1 ++; else c2 ++;

```

```

|

```

上述第(1)条语句使循环体重复执行 9 次,每次输出 i 的当前值和一个空格。

第(2)条语句省略了 <表达式 3>,并且循环体是一条空语句,该循环体被重复执行 1000 次,同时进行 1000 次  $i++ \leq 1000$  表达式的计算。

第(3)条语句中的 <表达式 1> 分别给 i 和 j 赋初值为 0,并对它们进行变量说明,<表达式 2> 和 <表达式 3> 分别为关系表达式和逗号表达式,循环体是一条赋值语句。

第(4)条语句中省略了全部三个表达式,循环体是一条复合语句。

第(5)条语句中的 <表达式 1> 为逗号表达式,循环体是一条复合语句,该循环语句完成从键盘上输入 n 个常数,并把它们依次累加到 y 上的任务。

第(6)条语句中的循环体是一个条件语句,它将被反复执行,直到  $k < \sqrt{m}$  不成立时为止。

第(7)条语句中省略了 <表达式 1>,<表达式 2> 为一个简单变量 b,<表达式 3> 是一个逗号表达式,循环体是一条赋值语句。

第(8)条语句的循环体将被循环执行 20 次,每次首先得到 0~99 之间的一个随机数 a 并输出它,接着若 a 为奇数就使 c1 增 1,否则使 c2 增 1。该循环的功能是得到并输出 0~99 之间的 20 个随机数,并分别统计出奇数和偶数的个数。

在 for 循环的循环体中允许使用 break 语句,其作用是:当执行到该语句时,就使执行流程转出所属的 for 循环语句,然后再向下顺序执行。

#### 4. 语句嵌套

for 循环体可以为任何可执行语句,当然也可以直接为一条 for 语句,或者在作为循环体的复合语句内使用 for 语句,并且嵌套的层数不受限制。如:

```

(1) for(i = 1; i <= 5; i++)
    for(j = 1; j <= 6; j++) s += i * j;
(2) for(i = 1; i <= 5; i++) {
    for(j = 1; j <= i; j++) cout << ' * ';
    cout << endl;
}
(3) for(i = 0; i < m; i++)
    for(j = 0; j < n; j++)
        if(aa[i][j] > max) {
            max = aa[i][j];
            row = i; col = j;
        }

```

以上每一条语句都是 for 双重循环语句,处于外面的称为外循环,内部的称为内循环。如对于第(1)条语句,外循环控制循环体(即内循环)执行 5 次,每次执行内循环时又控制内循环

体执行 6 次,所以内循环共被执行  $5 \times 6 = 30$  次。同理,第(2)条语句的内循环体(即 `cout << '*'`;语句)共被执行  $1+2+3+4+5=15$  次,第(3)条语句的内循环体(即 `if` 语句)共被执行  $m \times n$  次。

以上三条语句的执行流程分别如图 3-5(a),(b)和(c)所示。

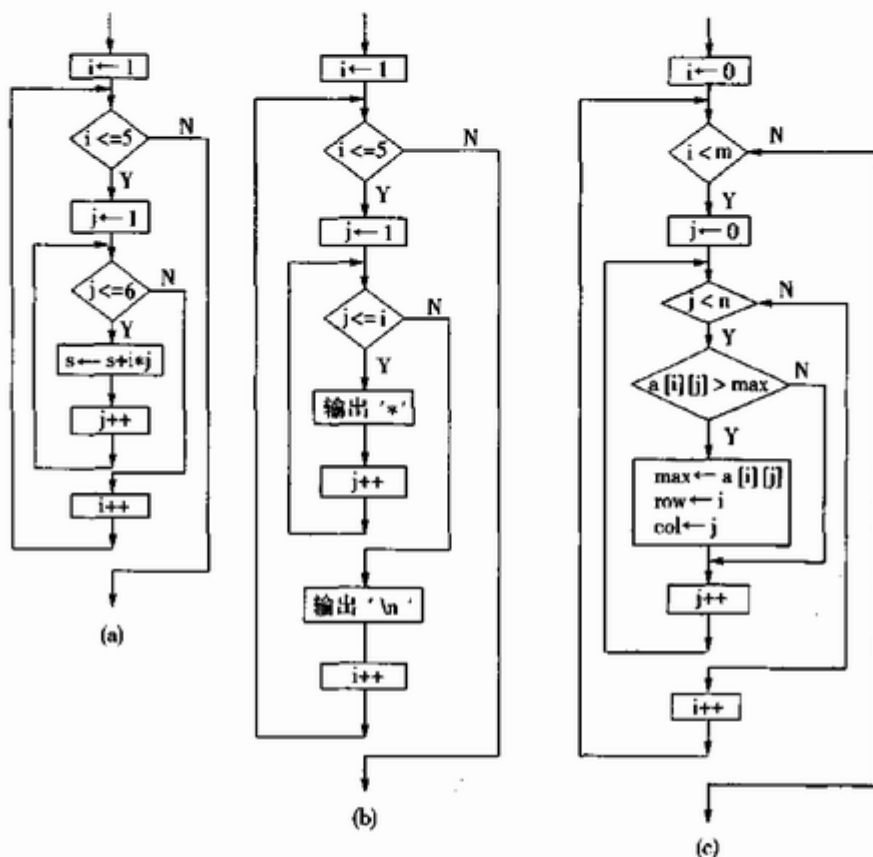


图 3-5 双重 for 循环语句的执行过程

## 5. 程序举例

```

(1) #include <iomanip.h>
void main()
{
    double x,y;
    cout << "从键盘上同一行输入 6 个常数: ";
    for(int i=0; i<6; i++) {
        cin >> x;
        y = 4 * x * x - 2 * x + 5;
        cout << "x = " << setw(5) << x;
        cout << setw(10) << "y = " << setw(5) << y << endl;
    }
}
  
```

在这个程序的主函数中,第1行定义了两个双精度变量  $x$  和  $y$ ,第2行给出提示信息,要求用户从键盘上输入6个常数后回车,第3~8行为一条 `for` 循环语句,其循环体是一条复合语句,将被重复执行6次,对应控制循环的变量  $i$  的取值依次为0、1、2、3、4和5,每次循环首先从键盘缓冲区读入一个常数并赋给  $x$ ,接着计算出  $y$  的值,然后按一定格式输出  $x$  和  $y$  的值。若程序运行后,从键盘上输入的6个常数为2、5、10、3.6、8.25、24.66,则得到的输出结果为:

从键盘上同一行输入6个常数:2 5 10 3.6 8.25 24.66

```
x= 2      y= 17
x= 5      y= 95
x= 10     y= 385
x= 3.6    y= 49.64
x= 8.25   y= 260.75
x= 24.66  y= 2388.14
```

```
(2) #include <iostream.h>
void main()
{
    int i,n; double p=1;
    cout << "输入一个正整数,求其阶乘: ";
    cin >> n;
    for(i=1; i <= n; i++) p*=i;
    cout << n << "!=" << p << endl;
}
```

在这个程序中定义了3个变量,用  $i$  作为控制循环的变量,简称循环变量,用  $n$  保存从键盘输入的一个正整数,用  $p$  计算和保存  $n$  的阶乘值, $p$  的初值为1,每次进行循环计算时都使  $p$  累乘循环变量  $i$  的值,循环结束后  $p$  的值就是  $n$  的阶乘值。假定程序运行时输入的  $n$  值为10,则运行结果为:

输入一个正整数,求其阶乘:10  
10 != 3.6288e+006

```
(3) #include <iostream.h>
#include <stdlib.h>
void main()
{
    int n,x,max,min;
    cout << "输入待处理数据的个数: ";
    cin >> n;
    if(n <= 0) {cout << "n <= 0!" << endl; exit(1);}
    cout << "输入 " << n << " 个待处理的数据: ";
    cin >> x; max=min=x;
    for(--n; n>0; n--) {
        cin >> x;
        if(x>max) max=x;
        if(x<min) min=x;
    }
    cout << "max: " << max << endl;
    cout << "min: " << min << endl;
}
```



}

在程序的主函数中,第1行同时定义了4个整型变量  $n, x, \max$  和  $\min$ ,用它们分别保存待处理数据的个数、当前被处理的一个数据、已处理数据中的最大值和已处理数据中的最小值。第2行和第3行用来从键盘上给  $n$  输入一个整数。第4行用来处理数据个数  $n$  小于等于0的不正常情况。第5行给出请用户输入  $n$  个数据的提示信息。第6行用来从键盘缓冲区读入第一个被处理数据并用它作为  $\max$  和  $\min$  这两个变量的初值。第7~11行为一个  $\text{for}$  循环,循环体共需执行  $n-1$  次,每次首先从键盘缓冲区读入一个数据到  $x$  中,接着分别同当前最大值  $\max$  和当前最小值  $\min$  相比较,若  $x$  较大则用它修改  $\max$  的值,若  $x$  较小则用它修改  $\min$  的值,使  $\max$  和  $\min$  始终保持已处理数据中的最大值和最小值,当此循环结束后,  $\max$  和  $\min$  中就分别存有  $n$  个数据中的最大值和最小值。最后两行语句输出所求得的最大值和最小值。

假定需处理6个数据,这6个数据为:48, 62, 30, 24, 55, 36,则程序运行结果为:

```
输入待处理数据的个数:6
输入6个待处理的数据:48 62 30 24 55 36
max:62
min:24
```

```
(4) #include<iostream.h>
void main()
{
    for(int a=0,b=1;b<100;){
        cout<<a<<' '<<b<<' ';
        a=a+b;
        b=a+b;
    }
    cout<<endl;
    cout<<a<<' '<<b<<' '<<endl;
}
```

该程序的主函数中包含有一个  $\text{for}$  循环,<表达式1>分别给变量  $a$  赋初值为0和给变量  $b$  赋初值为1,由于这两个变量在此之前没有被定义,所以在此使用时必须定义,<表达式2>是一个关系表达式  $b<100$ ,<表达式3>被省略,循环体中首先输出  $a$  和  $b$  的值,接着根据  $a$  和  $b$  的当前值求出  $a$  的新值,再利用  $a$  和  $b$  的当前值求出  $b$  的新值,然后判断  $b<100$  是否成立,若是则执行下一次循环,否则结束循环,转去执行后面的输出语句。该程序的运行结果为:

```
0 1 1 2 3 5 8 13 21 34 55 89
144 233
```

该程序的功能是输出一个数列的前若干项,其中第一项为0,第二项为1,以后每一项等于其前两项之和。如第10项为34,它等于第8项13和第9项21之和。

```
(5) #include<iostream.h>
const int M=4, N=5;
void main()
{
```

```

int i, j, s = 0;
for(i = 1; i <= M; i++)
    for(j = 1; j <= N; j++)
        s += i * j;
cout << s << endl;
}

```

主函数中使用了一个双重 for 循环,外循环变量  $i$  初值为 1,终值为整数常量  $M$ ,每执行一次外循环体(即内循环)后其值增加 1,内循环变量初值为 1,终值为整数常量  $N$ ,每执行一次内循环体(即  $s += i * j$ ; 语句)后其值也增加 1,内循环体共需执行  $M \times N$  次。该程序的功能是计算  $\sum_{i=1}^M \sum_{j=1}^N (i * j)$  的值。程序运行结果为 150。

## 6. 应用举例

例 1. 编一程序计算  $1 + 2^2 + 4^2 + 6^2 + \dots + 50^2$  的值。

分析:此题所给的计算公式是一个和式,它除第一项外,其余项为从 2 ~ 50 的每一个偶数的平方,因此可采用循环累加的方法来计算,即依次把每个数据项(在此为偶数的平方)累加到一个变量中。设循环变量为  $i$ ,它的初值、终值和步长(即每次循环后循环变量的增加值)应分别为 2、50 和 2,设用于累加的变量为  $s$ ,它的初值应为和式中的第一项 1,因为它不能够通过有规律的循环累加到  $s$  上。在循环体中通过赋值语句每次把  $i$  的平方值累加到  $s$  上,当循环结束后, $s$  的值就是所求的结果。根据分析编写出程序如下:

```

#include <iostream.h>
void main()
{
    int i, s = 1;
    for(i = 2; i <= 50; i += 2) s += i * i;
    cout << "s = " << s << endl;
}

```

例 2. 编一程序计算  $\sum_{i=1}^{10} (-1)^{i+1} \frac{x^i}{i!}$  的值,其中  $x$  值由键盘输入。

分析:此题是一个累加求和问题,适合使用 for 循环来实现。设循环变量为和式中的  $i$ ,它从 1 取值到 10,每次增长 1,每次计算出一个数据项并把它累加起来。为了计算一个数据项中的  $x^i$  和  $i!$ ,还需要设定两个累乘变量,假定分别用  $p1$  和  $p2$  表示,它们的初值应均为 1,在循环体中需要分别向  $p1$  和  $p2$  累乘  $x$  和  $i$  的值。为了把每个数据项的值累加起来,需要设定一个累加变量,假定用  $s$  表示,它的初值为 0,每次向它累加  $(-1)^{i+1} p1/p2$  的值。当和式中的所有 10 个数据项都累加到  $s$  之后, $s$  的值就是所求的结果。根据分析编写出程序如下:

```

#include <iostream.h>
void main()
{
    double x, p1 = 1, p2 = 1, s = 0;
    int i, j = 1;
    cout << "输入 x 的值: ";
    cin >> x;
    for(i = 1; i <= 10; i++) {

```

```

        p1 *= x;          // p1 的值为  $x^i$ 
        p2 *= i;          // p2 的值为  $i!$ 
        s += j * p1 / p2;  // j 的值为  $(-1)^{i+1}$ 
        j = -j;           // j 取反, 为下一数据项计算做准备
    }
    cout << s << endl;
}

```

例 3. 已知  $y_1 = \frac{1+e^x}{\sqrt{2a+a}}$ ,  $y_2 = \frac{1+e^{-x}}{\sqrt{2a-1}}$ , 其中  $x$  从 0 开始取值, 每次增加的步长为 0.25, 直到 3,  $a$  的值由键盘输入, 并要求大于 0, 编一程序依次求出  $x$  每一取值所对应的  $y_1$  和  $y_2$  的值。

分析: 设  $i$  为循环变量, 让它的初值、终值和步长分别为 0, 12 和 1, 则  $x$  的每次取值可表示为  $0.25i$ 。在循环体中计算  $y_1$  和  $y_2$  的公式应分别表示为:

```

y1 = (1 + exp(x)) / (sqrt(2 * a) + 1);
y2 = (1 + exp(-x)) / (sqrt(2 * a) - 1);

```

每次根据  $x$  的值(即  $0.25i$ )求出对应的  $y_1$  和  $y_2$  后都要输出出来。根据分析编写出程序程序如下:

```

#include <iomanip.h>
#include <math.h>
void main()
{
    double x, a, y1, y2;
    cout << "Input a(a>0): "; cin >> a; //也可把 a 设定为数值常量
    for(int i = 0; i <= 12; i++) {
        x = 0.25 * i;
        y1 = (1 + exp(x)) / (sqrt(2 * a) + 1);
        y2 = (1 + exp(-x)) / (sqrt(2 * a) - 1);
        cout << setw(10) << x << setw(10) << y1 << setw(10) << y2 << endl;
    }
}

```

例 4. 已知一组实验数据: 3.62, 2.93, 3.16, 3.73, 2.86, 3.40, 2.86, 3.07, 3.29, 3.24, 编一程序分别求出它们的平均值、方差和均方差, 要求每一结果只保留两位小数。

分析: 设它们的平均值、方差和均方差分别用变量  $v$ ,  $f$  和  $t$  表示, 由数学知识可知, 相应的计算公式为:

$$v = \frac{1}{n} \sum_{i=1}^n x_i \quad f = \frac{1}{n} \sum_{i=1}^n x_i^2 - v^2 \quad t = \sqrt{f}$$

其中  $n$  表示数据个数,  $x_i$  表示第  $i$  个数据。

此题需要首先求出  $\sum_{i=1}^n x_i$  和  $\sum_{i=1}^n x_i^2$ , 然后才能够求出  $v$ ,  $f$  和  $t$ 。而求所有数之和以及求所有数平方之和需要采用循环累加的方法。为此设循环变量为  $i$ , 它的初值、终值和步长应分别为 1,  $n$  和 1, 设输入变量为  $x$ , 每次从键盘缓冲区得到一个实验数据, 设累加数据之和的变量为  $s1$ , 累加数据平方之和的变量为  $s2$ 。每次分别向  $s1$  和  $s2$  累加  $x_i$  和  $x_i^2$  的值。根据以上分析编写出程序如下:

```

#include <iostream.h>
#include <math.h>
const int n=10;    // n 等于待处理数据的个数
void main()
{
    double x,s1,s2;
    s1=s2=0;
    cout << "从键盘上输入 "<<n<< "个实验数据:";
    for(int i=1; i <=n; i++){
        cin >> x;
        s1 += x;
        s2 += x * x;
    }
    double v,f,t;
    v=s1/n;
    f=s2/n-v*v;
    t=sqrt(f);
    v=floor(v*100)/100;
    f=floor(f*100)/100;
    t=floor(t*100)/100;
    cout << "v= " <<v << endl;
    cout << "f= " <<f << endl;
    cout << "t= " <<t << endl;
}

```

该程序上机运行后,按所给数据输入,则运行结果为:

```

从键盘上输入 10 个实验数据:
3.62 2.93 3.16 3.73 2.86 3.40 2.86 3.07 3.29 3.24
v=3.21
f=0.08
t=0.28

```

例 5. 由勾股定理可知,在一个直角三角形中,两条直角边  $a$  和  $b$  与斜边  $c$  的关系为  $a^2 + b^2 = c^2$ , 编一程序求出每条直角边均不大于 30 的所有整数解。如(3,4,5), (5,12,13)等都是该题的解。

分析:根据题意,需要使用二重循环来解决,设外循环变量用  $a$  表示,它的初值、终值和步长应分别取 1,30 和 1,内循环变量用  $b$  表示,它的初值、终值和步长应分别取  $a+1$ ,30 和 1。内循环变量的初值若取 1,而不是取  $a+1$ ,则会出现像(3,4,5)和(4,3,5)这样的重复组,为了避免重复组的出现,所以让  $b$  从  $a+1$  开始,即使第二条直角边大于第一条直角边。根据分析编写出程序如下:

```

#include <iostream.h>
#include <math.h>
const n=30;
void main()
{
    int a,b;
    double c;
    for(a=1; a <=n; a++)

```

```

        for(b = a + 1; b <= n; b++) {
            c = sqrt(a * a + b * b);    // 求出斜边的长度
            if(floor(c) == c)          // 若斜边同为整数则输出
                cout << '(' << a << ', ' << b << ', ' << c << ')' << endl;
        }
    }
}

```

该程序运行后,将得到如下输出结果:

```

(3,4,5)
(5,12,13)
(6,8,10)
(7,24,25)
(8,15,17)
(9,12,15)
(10,24,26)
(12,16,20)
(15,20,25)
(16,30,34)
(18,24,30)
(20,21,29)
(21,28,35)

```

例 6. 编一程序打印出 2~99 之间的所有素数。

分析:由数学知识可知,若一个自然数是素数(又称质数),则它必定不能被 1 和它本身之外的任何自然数整除。因为任何一个自然数都不可能比它大的自然数整除,所以要判断一个自然数是否为素数,只要看它能否被比它小的自然数(当然除 1 之外)整数,若能则不是素数,否则是素数。另一方面,若一个自然数  $n$  不是素数,则必然能表示成两个自然数  $n_1$  和  $n_2$  之积,并且若  $n_1$  小于等于  $\sqrt{n}$ ,  $n_2$  必然大于等于  $\sqrt{n}$ 。所以要判断一个自然数  $n$  是否为素数,可简化为判断它能否被 2 至  $\sqrt{n}$  之间的自然数整除即可。因为若一个自然数  $n$  不能被 2 至  $\sqrt{n}$  之间的自然数整除,则必然也不能被  $\sqrt{n}$  至  $n-1$  之间的自然数整除。

由以上分析可知,判断一个自然数  $n$  是否为素数的过程是一个循环过程,设循环变量为  $i$ ,它的初值、终值和步长应分别为 2,  $\text{floor}(\sqrt{n})$  和 1,在循环体内要判断  $n$  是否能被  $i$  整除,若能则表明  $n$  不是素数,应结束循环,若不能则继续循环。当整个循环正常结束(即因 <表达式 2> 的值为 0 而结束循环的情况)后,表明  $n$  不能被 2 至  $\sqrt{n}$  之间的任何自然数整除,得到  $n$  是一个素数。

要求出所给的 2~99 区间内的所有素数,需要依次对每个整数进行判断,这又是一个循环处理的过程。为此设循环变量为  $n$ ,它的初值、终值和步长应分别为 2,99 和 1,对于  $n$  的每一取值,都要执行判断它是否为素数的循环过程,所以解决此题的程序模块结构是一个双重循环。

根据以上分析,编写出程序如下:

```

#include <iostream.h>
#include <math.h>
void main()
{
    int i,n;

```

```

    for(n=2; n <= 99; n++) {
        int temp = (int)floor(sqrt(n));
        for(i=2; i <= temp; i++)
            if(n%i == 0) break;    // 执行 break 时为非正常结束循环
        if(i > temp) cout << n << ' ';
    }
    cout << '\n';
}

```

若这个程序中的 for 内循环执行结束后,若 i 的值大于 temp,则表明内循环是正常结束的, n 为一个素数,所有要把它打印出来,否则内循环是非正常结束的, n 是一个非素数,此时的 i 值必然小于等于 temp,它不会被打印出来。

该程序运行后得到的输出结果为:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

### 3.4 while 语句

#### 1. 语句格式

while 语句又称为 while 循环,它也是一种结构性语句,它的循环体是一条语句。while 语句格式为:

```
while(<表达式>) <语句>
```

<语句> 成分是 while 语句的循环体,它可以是任何一条可执行语句或空语句。

#### 2. 执行过程

while 语句的执行过程为:

- (1) 计算 <表达式> 的值,假定为 M;
- (2) 若 M 为非 0,则执行一遍循环体,否则结束整个语句的执行;
- (3) 自动转向第(1)步执行。

While 语句的执行过程可用图 3-6 表示出来。

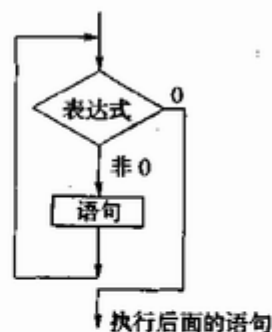


图 3-6 while 语句的执行过程

#### 3. 格式举例

- (1) while(x <= 0) cin >> x;
- (2) while(x) {s += x; cin >> x;}
- (3) while (n -- ) {
 cin >> x;
 if(x > 0) n1 ++; else n2 ++;
 }
- (4) while(i < n && x != a[i]) i ++;

```

(5) while(i++ < N) {
    x = rand() % 100;
    if(x%2 == 0) c2 ++;
    if(x%3 == 0) c3 ++;
    if(x%5 == 0) c5 ++;
}

(6) while(1) {
    cout << "输入一个运算符(+, -, *, /或@): ";
    cin >> op;
    if(op == '@') break;
    switch(op) {
        case '+': z = Add(x, y); break;
        case '-': z = Subt(x, y); break;
        case '*': z = Mult(x, y); break;
        case '/': z = Divide(x, y); break;
        default: cout << "Input error! " << endl;
    }
}

```

对于每一条 while 语句,若第一次计算 < 表达式 > 的值为 0,则循环体不会被执行就离开了循环,否则循环体至少被执行一次。

在 while 语句的循环体内,也可以同在 for 语句的循环体内一样使用 break 语句,使之非正常地结束其执行过程,转向所属 while 语句的后面继续向下执行。

请读者分析以上每一条 while 语句的执行过程。

while 循环中的循环体语句可以为任何一条可执行语句或空语句,因此同样可以为一条 while 语句或其他循环语句,若循环体是一条复合语句,则在复合语句内也同样可以使用 while 语句或其他循环语句。总之,允许各种循环语句之间的嵌套使用,并且嵌套的层数不受限制。

#### 4. 程序举例

```

(1) #include <iostream.h>
void main()
{
    int x, c1 = 0, c2 = 0;
    cin >> x;
    while(x >= 0) { //当输入一个负数时结束循环
        if(x < 60) c1 ++; else c2 ++;
        cin >> x;
    }
    cout << c1 << ' ' << c2 << endl;
}

```

该程序的功能是:分别统计出从键盘上输入的所有整数中小于 60 和大于等于 60 的数



据个数,然后显示出来。在程序中用输入负数作为终止 while 循环的结束标志,使用 x 作为输入变量,使用 c1 和 c2 作为统计变量。

```
(2) #include<iostream.h>
void main()
{
    int a,b;
    cout << "请输入两个正整数: ";
    cin >> a >> b;
    while(a <= 0 || b <= 0) {cout << "重新输入: "; cin >> a >> b;}
    while(b) {
        int r = a % b;
        a = b; b = r;
    }
    cout << a << endl;
}
```

该程序的功能是:采用辗转相除法求出两个整数的最大公约数。

如假定从键盘上输入的两个整数为 136 和 40,用它们分别作为 a 和 b 的值,因 b=40 不为 0,所以执行第一遍 while 循环体,使得 r 为 a 除以 b 而得到的余数,接着把 a 和 b 修改为除数 b 和余数 r 的值,即 40 和 16;又因 b 的当前值为 16,它不为 0,接着执行第二遍循环体,使得 r 的值为 8,接着把 a 和 b 修改为 16 和 8;再进行条件判断时,因 b=8 不为 0,接着执行第三遍循环体,使得 r 的值为 0,a 和 b 的值再一次被修改为 8 和 0;进行第四次 while 循环条件判断时,因 b 等于 0,所以结束循环。结束循环后 a 的值就是原有两个整数 136 和 40 的最大公约数。

利用辗转相除法求 136 和 40 的最大公约数的计算步骤为:

(1)  $40 \overline{)136} \dots 16$

3

(2)  $16 \overline{)40} \dots 8$

2

(3)  $8 \overline{)16} \dots 0$

2

最后一步中的除数 8 就是 136 和 40 的最大公约数。

```
#include<iostream.h>
#include<stdlib.h>
#include<math.h>
void main()
{
    int i=10,a;
    while(i>0) {
        a=rand() % 190 + 10;
        int j, k=int(sqrt(a));
        for(j=2; j<=k; j++)
            if(a % j == 0) break;
        if(j>k) {cout << a << ' '; i--;}
    }
}
```

}

该程序是一个双重循环,外层为 while 循环,内层为 for 循环,每执行一遍外循环体可能显示出一个 10~200 之间的一个素数。

该程序的功能是:随机产生出 10 个 10~200 之间的素数并显示出来。

## 5. 应用举例

例 1. 编一程序求出满足不等式  $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \geq 5$  的最小  $n$  值。

分析:此题不等式的左边是一个和式,该和式中的数据项个数是未知的,也正是要求出的。对于和式中的每个数据项,对应的通式为  $\frac{1}{i}, i = 1, 2, \dots, n$ ,所以可采用循环累加的方法来计算出不等式的和。设循环变量为  $i$ ,它应从 1 开始取值,每次增加 1,直到不等式的值不小于 5 为止,此时的  $i$  值就是所求的  $n$ 。设累加变量为  $s$ ,在循环体内应把  $1/i$  的值累加到  $s$  上。

根据以上分析,采用 while 循环编写出程序如下:

```
#include <iostream.h>
void main()
{
    int i = 0; double s = 0;
    while(s < 5) s += double(1)/++i;
    cout << "n = " << i << endl;
}
```

若采用 for 循环编写程序,则如下所示:

```
#include <iostream.h>
void main()
{
    int i; double s = 0;
    for(i = 1; s < 5; i++) s += double(1)/i;
    cout << "n = " << i - 1 << endl;
    // 注意: 此 i - 1 的值为所求的 n 值
}
```

该程序的输出结果应为:  $n = 83$ 。

例 2. 一家商场采用打折促销活动,具体做法是:购物满 100 元送 30 元购物券,用购物券购物同用人民币购物一样遵循上述原则。若一个顾客一次购物花销  $x$  元,则最终能够得到几折优惠。

分析:因购买每百元物品送 30 元购物券,不满百元部分将不赠送,所以花销  $x$  元应得到的购物券为  $(x/100) \times 30$ ,假定这个值仍利用  $x$  保存,则再购价值为  $x$  的物品后,同样又可以得到由上述公式计算出来的购物券,依次类推,直到  $x$  的当前值为 0 时止。

购物支付的金额与所购物品价值的比称为折或折价。如花销 70 元购买 100 元的物品则称为 7 折。

在此例中花销了  $x$  元,应购买到  $x_1 + x_2 + \dots + x_n$  元的物品,其中  $x_1$  等于初次购物的开支

$x_1, x_2 = (x_1/100) \times 30, x_3 = (x_2/100) \times 30, \dots$ , 直到  $x_{n+1}$  为 0 时止。设购买到物品的价值用  $s$  表示, 初次购物所花费的金额用变量  $a$  保留起来, 则购买物品的最终折价为  $a/s$ , 其中  $s = x_1 + x_2 + \dots + x_n$ 。

根据分析, 编写出程序如下:

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
void main()
{
    int x, a, s;
    cout << "请输入初次购物所花费的现金(元): ";
    cin >> x;
    if(x <= 0) exit(1);
    a = x; s = 0;
    while(x) {
        s += x;
        x = x/100 * 30;
    }
    cout << a << " " << s << " " << float(a)/s << endl;
}
```

程序运行后, 假定从键盘上输入的  $x$  值为 2650 元, 则得到的显示结果为:

```
请输入初次购物所花费的现金(元):2650
2650 3700 0.716216
```

例 3. 有一家企业, 若年产值平均增长率分别按 2%, 4%, 6%, ..., 20% 计算, 问分别需要经过多少年才能够使年产值翻一番。

分析: 假定把当年的年产值定为 1 个单位, 则翻一番后就应变为 2。设年产值平均增长率为  $x$ , 经过的年数为  $n$ ,  $n$  年后的产值为  $y$ , 则求  $y$  的计算公式为:

$$y = (1 + x)^n$$

由题意可知, 当  $y$  正好等于 2 或刚好超过 2 时所得到的  $n$  值就是按年平均增长率为  $x$ , 达到翻一番所需要的年数。要根据  $x$  值求出  $y$  达到 2 之后的  $n$  值, 应采用循环来解决。设循环变量为  $n$ , 它从 1 开始取值, 每次增加 1, 每次向累乘变量  $y$  (它的初值应为 1) 乘上  $1 + x$  的值, 当  $y < 2$  成立时继续下一次循环, 直到  $y \geq 2$  为止, 此时的  $n$  值就是所求的年数。

根据题目要求,  $x$  不是取一次值, 而是取多次值。对于  $x$  的每一次取值, 都需要求出对应的  $n$  值。由于  $x$  的取值是有规律的, 它从 0.02 开始到 0.20 结束, 每次增加 0.02, 所以可使用  $x$  作为 for 循环的循环变量, 控制循环体的循环执行的次数, 每次循环求出  $x$  值所对应的  $n$  值。

根据分析编写出程序如下:

```
#include <iostream.h>
void main()
{
    double x, y;
    int n;
```

```

for(x=0.02; x <= 0.20; x += 0.02) {
    n=0; y=1;
    while(y<2) {
        n++;
        y*=1+x;
    }
    cout << x*100 << "% " << n << ' ' << y << endl;
}
}

```

若上机运行该程序,则可得到如下的显示结果:

```

2% 36 2.03989
4% 18 2.02582
6% 12 2.0122
8% 10 2.15892
10% 8 2.14359
12% 7 2.21068
14% 6 2.19497
16% 5 2.10034
18% 5 2.28776
20% 4 2.0736

```

## 3.5 do 语句

### 1. 语句格式

do 语句又称 do 循环,它也是一种结构性语句,其语句格式为:

```
do <语句> while (<表达式>);
```

其中<语句>是 do 循环的循环体,它可以为任何可执行语句或空语句。

### 2. 执行过程

do 语句的执行过程为:

(1) 执行一遍循环体;

(2) 求作为循环条件使用的<表达式>的值,若其值非 0 则自动转向第(1)步,否则结束 do 循环的执行过程,继续执行其后面的语句。

图 3-7 所示描述了 do 循环的执行过程。

在 do 语句的循环体中,也可以使用 break 语句,用它来非正常结束该循环的执行,使执行流程转向所属 do 语句的后面。

### 3. 格式举例

(1) do i++; while(x[i] < y);

(2) do cin >> x; while(x <= 0);



图 3-7 do 语句执行流程

```

(3) do { cin >> x;
        s += x;
    } while( -- n > 0);
(4) do {
        int x = rand() % 98 + 2;
        int y = int(sqrt(x));
        for(int i = 2; i <= y; i++)
            if(x % i == 0) break;
        if(i > y) { n++; cout << x << " is prime. " << endl; }
    } while(n < 5);

```

第一条语句中的循环体执行  $i++$  的操作,当数组元素  $x[i]$  的值小于  $y$  时,转去执行下一遍循环体,直到条件  $x[i] < y$  不成立为止。

第二条语句中的循环体执行从键盘上输入一个数据的操作,当  $x \leq 0$  成立时,则重新给  $x$  输入数据,一旦输入的数据大于 0 则结束循环输入过程,继续向下执行。

第三条语句的功能是把从键盘上输入的  $n$  个数值累加到变量  $s$  中。其中  $n$  表示进入此循环前的  $n$  的值。

第四条语句的功能是连续求出并输出 5 个(假定  $n$  的初值为 0)随机产生的 2~99 之间的素数。在这条语句的循环体中又使用了  $\text{for}$  循环,从而构成了双重循环。

在 C++ 语言中,共包含有三种循环语句,到此全部介绍完了,其中  $\text{do}$  语句的循环体至少被执行一遍,其他两种语句的循环体可能一次都不会被执行。 $\text{do}$  循环称为先执行(循环体)后判断,其余两种语句称为先判断后执行(循环体)。另外,每一种循环语句内都可以嵌套任一种循环语句,并且嵌套的层数不受限制。

在实际编程中,对于重复计算或重复处理的问题,可以采用任一种循环语句编写,只要描述正确,从而能够得到正确的运行结果即可。

#### 4. 程序举例

```

(1) #include <iostream.h>
    const int NM = 10;
    void main() {
        int x, n = 1, c = 0;
        do {
            cin >> x;
            if(x >= 30 && x <= 60) c++;
        } while(n++ < NM);
        cout << "c = " << c << endl;
    }

```

该程序的功能是:接收从键盘上输入的  $NM$  个整数,统计出 30~60 范围内的整数个数,最后输出统计结果。

```

(2) #include <iostream.h>
    void main()
    {

```

```

int x;
cout << "请输入一个整数,若小于3则重输:";
do cin >> x; while(x <= 2);
int i = 2;
do{
    while(x%i == 0){
        cout << i << ' ';
        x/=i;
    }
    i++;
}while(i < x);
if(x != 1) cout << x;
cout << endl;
}

```

在这个程序中,第6行为do循环,它确保输入给x的是一个大于等于3的整数,第7行定义整数变量i并赋予2作为初值,第8~14行为一个do循环,循环体中的第一条语句为while循环,每当x能够被i整除就输出i的值和一个空格,接着修改x为除以i的整数商,第2条语句使i增1,每次执行完do循环体后,都判断条件i < x是否成立,若成立则进入下一轮循环,否则结束循环,接着执行后面的条件语句。

此程序的功能是:把从键盘上输入的一个大于等于3的整数分解为质因子的乘积。如输入24时得到的输出结果为“2 2 2 3”,输入50时得到的输出结果为“2 5 5”,输入37时得到的输出结果为“37”。

## 5. 应用举例

例1. 编一程序把从键盘上输入的一个十进制整数转换为对应的十六进制数字串输出。

分析:由计算机基础知识可知,一个十进制整数转换为任意r进制的整数时应采用逐次除r取余法。具体算法为:首先用待转换的十进制整数d整除以r得到余数,它就是对应r进制数的最低位,以后每次用上一次d整除以r的整数商作为被除数除以r,得到对应r进制数的较高位,最后商为0得到的余数是对应r进制数的最高位。如把十进制数74分别转换为二进制数和十六进制数时,对应的转换过程如图3-8(a)和(b)所示。

2	74	余数	
2	37	... 0	
2	18	... 1	
2	9	... 0	
2	4	... 1	
2	2	... 0	
2	1	... 0	
	0	... 1	
(a)			
			余数
16	74	... A	
16	4	... 4	
(b)			

图 3-8 十进制整数转换为r进制数的运算过程

由图 3-8 可知,十进制数 74 所对应的二进制数为 1001010,对应的十六进制数为 4A。

以上转换过程是一个重复处理的过程,适合采用循环来解决。按题目要求,每次循环用被除数  $x$  (开始为待转换的十进制整数) 整除以 16 所得余数赋给一个整数变量  $rem$ , 把所得到的整数商又赋给  $x$ , 当  $rem$  在 0~9 之间时直接输出它, 否则应输出它所对应的十六进制数字字符。此循环直到  $x$  为 0 时止。

注意: 上述输出得到的十六进制数是按从低位到高位次序排列的, 对它再按相反次序排列时才是所求的十六进制数。待以后学习了数组, 就可以利用数组顺序存储转换过程中依次得到的每个数字位, 转换结束后再按相反的次序输出数组内容即可得到正确的结果。

根据分析, 编写出程序如下:

```
#include <iostream.h>
void trans(int x)
{ // 此函数用于把十进制整数 x 转换为十六进制数字串输出
  int rem; // 用于保存余数
  do {
    rem = x % 16;
    x = x / 16;
    if (rem < 10) cout << rem;
    else switch (rem) {
      case 10: cout << 'A'; break;
      case 11: cout << 'B'; break;
      case 12: cout << 'C'; break;
      case 13: cout << 'D'; break;
      case 14: cout << 'E'; break;
      case 15: cout << 'F'; break;
    }
  } while (x != 0);
  cout << endl;
}
void main()
{
  int d;
  cout << "从键盘输入一个十进制正整数: ";
  cin >> d;
  trans(d);
}
```

若把 `trans` 函数改写如下:

```
void trans(int x)
{ // 此函数用于把十进制整数 x 转换为十六进制数字串输出
  char a[10];
  int i = 0, rem;
  do {
    rem = x % 16;
    x = x / 16;
    if (rem < 10) a[i] = 48 + rem; // '0' 字符的 ASCII 码为 48
    else switch (rem) { // 此开关语句可用 a[i] = 55 + rem; 语句代替
```

```

        case 10: a[i] = 'A'; break;
        case 11: a[i] = 'B'; break;
        case 12: a[i] = 'C'; break;
        case 13: a[i] = 'D'; break;
        case 14: a[i] = 'E'; break;
        case 15: a[i] = 'F'; break;
    }
    i++;
} while(x != 0);
while(i > 0) cout << a[--i];
cout << endl;
}

```

若看不懂此函数,可暂时留着,待学习过数组内容以后再回过头来仔细阅读。

当运行上述在 `trans` 函数中使用数组的程序时,若输入的十进制整数为 1234,则得到的输出结果为:

```

从键盘输入一个十进制正整数:1234
4D2

```

例 2. 编一程序利用牛顿法求解方程  $e^x + 3x - 2 = 0$  的根,要求两相邻近似根之差的绝对值不大于 0.001。

分析:由数学知识可知,若令  $f(x) = e^x + 3x - 2$ ,则  $f'(x) = e^x + 3$ ,用牛顿法求方程  $e^x + 3x - 2 = 0$  的近似根的叠代计算公式为:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (i \geq 0, x_0 \text{ 可为任意值})$$

利用这个叠代公式求出一个新的近似根后,都要判断  $|x_{i+1} - x_i| \leq 0.001$  是否成立,若成立则就把新的近似根  $x_{i+1}$  作为方程的根,否则继续求出下一个近似根,直到上述不等式满足为止。

由上面叠代公式可知,新的近似根  $x_{i+1}$  只与刚求出的近似根  $x_i$  有关,而与其他已求出的近似根无关,同样,判断  $|x_{i+1} - x_i| \leq 0.001$  也只需要用到新的近似根和刚求出的近似根。所以可设置两个变量,假定分别为  $x1$  和  $x2$ ,用  $x2$  保存新的近似根,用  $x1$  保存刚求出的近似根。在每次计算新的近似根前,都要把  $x2$  的值赋给  $x1$ ,然后再根据公式  $x2 = x1 - f(x1)/f'(x1)$  计算出新的近似根  $x2$ 。

根据上述分析,编写出程序如下:

```

#include <iostream.h>
#include <math.h>
double func(double x)
{ // x 可以为任何值
    double x1, x2, y1, y2;
    x2 = x; // 给 x2 赋初值为 x
    do {
        x1 = x2;
        y1 = exp(x1) + 3 * x1 - 2; // y1 = f(x1)
        y2 = exp(x1) + 3; // y2 = f'(x1)
        x2 = x1 - y1/y2;
    } while (fabs(x2 - x1) > 0.001);
    return x2;
}

```



```

        }while(fabs(x2 - x1) > 0.001);
        return x2;
    }
    void main()
    {
        double x;
        cout << "从键盘输入任一实数作为自变量 x 的初值: ";
        cin >> x;
        x = int(func(x) * 1000)/1000.0; // 保留运算结果的 3 位小数
        cout << "root: " << x << endl;
    }

```

当这个程序运行后,从键盘上输入任一实数时,得到的输出结果为 "root:0.242"。

### 3.6 跳转语句

跳转类语句包括 goto、continue、break 和 return 四种语句。每一种语句都有规定的语句格式和作用。

#### 1. goto 语句

goto 语句称为无条件转向语句,其语句格式为:

```
goto < 语句标号 >;
```

< 语句标号 > 是一个用户命名的标识符,它必须同时出现在该 goto 语句所在函数的某一条语句的前面,并且它同该语句之间必须用冒号分开(当然在冒号前后可以使用任意多个空白符),用此标识符来标识该语句的开始位置,以便在其他地方使用的 goto 语句转向此位置。

当程序执行到 goto 语句时,将使执行流程转向 < 语句标号 > 所标识的位置,接着将从这个位置开始向后执行。

例如:有一个函数定义为:

```

void func(double x)
{
    double y;
    if(x < 0) {y = 3 * x * x - 1; goto finish;}
    if(x >= 0 && x <= 10) {y = exp(x)/3 + 2; goto finish;}
    y = 5 * sqrt(x) - 2 * x + 1;
finish:
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
}

```

在这个函数中使用了两条 goto 语句,它们带有相同的语句标号 finish,当执行到它们中任意一条语句时,都将转向标识符 finish 所标识的位置,接着从该位置起向下执行语句。

该函数的功能是:根据参数 x 的值计算并输出 y 的值,当 x 小于 0 时,计算 y 的公式为

$3x^2 - 1$ , 当  $x$  大于等于 0 同时小于等于 10 时, 计算  $y$  的公式为  $\frac{1}{3}e^x + 2$ , 当  $x$  大于 10 时, 计算  $y$  的公式为  $5\sqrt{x} - 2x + 1$ 。

在上述函数中使用 `goto` 语句完全是为了体验 `goto` 语句的功能, 实际上不是必须的, 完全可以编写出比它更简明易读的不使用 `goto` 语句的函数, 如下所示:

```
void func(double x)
{
    double y;
    if(x < 0) y = 3 * x * x - 1;
    else if(x >= 0 && x <= 10) y = exp(x)/3 + 2;
    else y = 5 * sqrt(x) - 2 * x + 1;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
}
```

在函数中使用 `goto` 语句转来转去, 破坏了程序自上而下顺序执行的次序, 不符合结构化程序设计思想, 所以应当尽量避免使用。但在特殊的场合, 如需要从多重循环的内部一次退到最外层循环的后面时, 使用 `goto` 语句是简单可行的。

下面的两个函数都是用来计算  $1 + 2^2 + 3^2 + \dots + 100^2$  的值并返回。第一个函数使用 `goto` 语句实现累加的循环计算, 第二个函数使用 `for` 语句实现累加的循环计算。由于第二个函数采用循环语句而非 `goto` 语句处理重复计算问题, 体现了结构化的编程方法, 而第一个函数的编程方法是非结构化的, 是应该放弃的方法。

```
int sum1() {
    int s = 0, i = 1;
start:
    s += i * i;
    i++;
    if(i <= 100) goto start;
    return s;
}

int sum2() {
    int s = 0;
    for(int i = 1; i <= 100; i++) s += i * i;
    return s;
}
```

## 2. continue 语句

该语句称为继续语句, 它只有语句关键字 `continue`, 没有其他任何成分。

该语句被限定使用在任一种循环语句的循环体中, 当程序运行时执行到该语句时, 将立即结束一次循环体的执行, 接着执行其后面的循环操作。

**注意:** `continue` 同 `break` 不一样, 执行 `break` 时是退出整个循环, 接着执行该循环后面的语句, 执行 `continue` 时是退出循环体的一次执行, 接着执行下一次循环操作。

下面的三个程序段具有完全相同的功能, 都是把 1 ~ 10 之间所有奇数的平方累加到变量 `sum` 中。

量  $s$  上。

```
(1) int i,s=0;
    for(i=1;i <= 10;i++){
        if(i%2 == 0) continue;
        s += i*i;
    }

(2) int i=0,s=0;
    while(++i <= 10) {
        if(i%2 == 0) continue;
        s += i*i;
    }

(3) int i=1,s=0;
    do {
        if(i%2 == 0) continue;
        s += i*i;
    }while(++i <= 10);
```

### 3. break 语句

该语句称为中断语句,它也只有语句关键字 `break`,没有其他任何成分。

该语句被限定使用在任一种循环语句和 `switch` 语句中,当程序执行到该语句时,将立即结束所在循环语句或 `switch` 语句的执行,接着执行其后面的语句。

**注意:** 当 `break` 语句出现在内层的循环语句或 `switch` 语句中时,它只是结束该内层的循环语句或 `switch` 语句的执行,不会结束其他外层循环或 `switch` 语句的执行。总之,`break` 只结束本层循环或 `switch` 语句的执行。

例如:

```
#include <iostream.h>
const int N=4,M=4,P=4;
void main()
{
    int i,j;
    for(i=1;i <= N;i++){
        for(j=1;j <= M;j++){
            if((i-j)*(i-j) == P) break;
            cout << i << ' ' << j << ' ';
            if(i == 3) break;
            cout << i*i+j*j << endl;
        }
        cout << endl;
    }
}
```

该程序的运行结果如下:

```
1 3 10
2 4 20
3 1
```

若把程序中的 `if(i == 3) break;` 语句改写为:

```
if(i == 3) {cout << endl; continue;}
```

则得到的运行结果如下:

```
1 3 10
2 4 20
3 1
4 2 20
```

#### 4. return 语句

该语句称为返回语句,其语句格式为:

```
return [<表达式>];
```

若该语句使用在类型为 `void` 的函数中,则它不能带有 `<表达式>` 选项,若使用在其他任何类型的函数中,则必须带有 `<表达式>` 选项。

在一个函数中可以使用一条或多条 `return` 语句,特殊地,允许在具有 `void` 类型的函数中不使用 `return` 语句。

当程序执行到 `return` 语句时,若它带有 `<表达式>`,则首先计算出它的值,然后把这个值作为整个函数的值返回到调用该函数的位置,若它不带有 `<表达式>`,同样使执行流程返回到调用该函数的位置,但不带回任何值。

若执行到一个函数结束仍碰不到 `return` 语句,则也将自动返回到调用该函数的位置,接着将从这个位置起向下执行。

请看下面的程序:

```
#include <iostream.h>
int f1(int n) {
    int s=0;
    for(int i=1;i<n;i+=2) s += i*i;
    return s*s;
}
void f2(int x) {
    cout << x << endl;
}
void main()
{
    int a;
    a = f1(6);
    f2(a);
    f2(f1(8));
}
```

当这个程序运行时,将依次执行主函数中的每条语句,当执行到第二条时,首先调用 `f1` 函数,把实参 6 传送给形参 `n`,接着执行 `f1` 函数体,执行到 `return` 语句时计算出 `s*s` 的值(即 1225)作为整个函数的值返回,返回后把返回值 1225 赋给变量 `a`;执行第三条语句时调用 `f2` 函数,把实参 `a` 的值 1225 传送给形参 `x`,执行该函数的函数体时打印出 `x` 的值,函数体执行

结束后自动返回主函数,表明第三条语句执行结束;执行第四条语句时,首先调用 f1 函数,返回的函数值为 7056,接着调用 f2 函数,打印出传送给该函数的实参值。程序运行结果为:

1225

7056

## 习题三

### (一) 填空题

1. 流程控制类语句包括\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_三类语句。
2. 跳转类语句包括\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_四种。
3. 选择类和循环类中的每一种语句都属于\_\_\_\_\_语句。
4. 在 switch 语句中,每个语句标号所含关键字 case 后面的表达式必须是\_\_\_\_\_。
5. 在嵌套的 if 语句中,每个 else 关键字与它前面最近的\_\_\_\_\_关键字相配套。
6. 作为语句标号使用的 C++ 保留字 case 和 default 只能用于\_\_\_\_\_语句中。
7. 执行 switch 语句时,在进行作为条件的表达式求值后,将从某个标号位置起向下执行,当碰到下一个标号位置时(停止/不停止)\_\_\_\_\_执行。
8. 任何\_\_\_\_\_语句都可以改写为具有同样功能的 if 语句来实现。
9. 在 for 语句中,假定循环体被执行次数为 n,则 <表达式 1> 共被计算\_\_\_\_\_次,<表达式 2> 共被计算\_\_\_\_\_次,<表达式 3> 共被计算\_\_\_\_\_次。
10. 执行 for 和 while 循环时,每次是先进行\_\_\_\_\_的判断,然后再执行\_\_\_\_\_,执行 do 循环时则相反。
11. continue 语句只能使用在\_\_\_\_\_类语句中,break 语句既可以使用在\_\_\_\_\_类语句中,也可以使用在\_\_\_\_\_语句中。
12. for 语句中 <表达式 2> 是在每次执行\_\_\_\_\_之前被计算,而 <表达式 3> 是在每次执行\_\_\_\_\_之后被计算。
13. 在所有结构性语句中,只有\_\_\_\_\_语句的最后必定是以分号结束的,而其余语句的最后一个字符可能是分号,也可能是\_\_\_\_\_。
14. \_\_\_\_\_语句的循环体至少被执行一次,\_\_\_\_\_和\_\_\_\_\_语句的循环体可能不会被执行。
15. 若 for 循环的“头”为 “for(int i=0; i<10; i++)”,并且在循环体中不会修改 i 的值,则循环体将被重复执行\_\_\_\_\_次后正常结束。
16. 若 while 循环的“头”为 “while(i++ <= 10)”,并且 i 的初值为 0,同时在循环体中不会修改 i 的值,则循环体将被重复执行\_\_\_\_\_次后正常结束。
17. 若 do 循环的“尾”为 “while(++i < 10)”,并且 i 的初值为 0,同时在循环体中不会修改 i 的值,则循环体将被重复执行\_\_\_\_\_次后正常结束。
18. 当在程序中执行到\_\_\_\_\_语句时,将结束本层循环类语句或 switch 语句的执行。

19. 当在程序中执行到\_\_\_\_\_语句时,将结束所在循环语句中循环体的一次执行。
20. 在程序中执行到\_\_\_\_\_语句时,将结束所在函数的执行过程,返回到调用该函数的位置。

(二) 写出下列每个程序运行后的输出结果并上机验证

```
1. #include <iostream.h>
void main()
{
    int a=2,b=5,c=4;
    if(a+b>10) c=a*b; else c=3*a+b;
    if(c<=20) cout<<c*c; else cout<<4+c-5;
    cout<<endl;
    a=a+b; b=a+b; c=a+b+c;
    cout<<"a,b,c="<<a<<','<<b<<','<<c<<endl;
}
```

```
2. #include <iostream.h>
void main()
{
    int x=5;
    switch(2*x-3) {
        case 4: cout<<x<<' ';
        case 7: cout<<2*x+1<<' ';
        case 10: cout<<3*x-1<<' '; break;
        default: cout<<"default"<<endl;
    }
    cout<<"switch end."<<endl;
}
```

```
3. #include <iomanip.h>
#include <math.h>
void main()
{
    int i,x,y;
    for(i=0; i<6; i++) {
        cin>>x;
        if(x<0) y=1;
        else if(x<10) y=x*x+3;
        else if(x<60) y=4*x-5;
        else y=int(sqrt(x));
        cout<<setw(5)<<x<<setw(5)<<y<<endl;
    }
}
```

假定从键盘上输入的6个常数为:36, -5, 73, 192, 6, 44。

```
4. #include <iostream.h>
void main()
{
    int s0,s1,s2,x;
```

```

s0 = s1 = s2 = 0;
for(int i = 0; i < 12; i++) {
    cin >> x;
    switch(x % 3) {
        case 0: s0 += x; break;
        case 1: s1 += x; break;
        case 2: s2 += x; break;
    }
}
cout << s0 << ' ' << s1 << ' ' << s2 << endl;
}

```

假定从键盘上输入的 12 个整数为:36,25,20,43,12,70,66,34,28,15,32,55。

5. #include <iomanip.h>  
const int N=5;  
void main()  
{  
int i,p=1,s=0;  
for(i=1;i<N;i++){  
p=p\*i;  
s=s+p;  
cout << setw(5) << i << setw(5) << p;  
cout << setw(5) << s << endl;  
}  
}
6. #include <iomanip.h>  
const int M=20;  
void main()  
{  
int c2,c3,c5;  
c2=c3=c5=0;  
for(int i=1;i<=M;i++){  
if(i%2==0) c2++;  
if(i%3==0) c3++;  
if(i%5==0) c5++;  
}  
cout << c2 << ' ' << c3 << ' ' << c5 << endl;  
}
7. #include <iomanip.h>  
void main()  
{  
int i,j;  
for(i=0;i<5;i++){  
for(j=i;j<5;j++) cout << '\*';  
cout << endl;  
}  
}
8. #include <iostream.h>

```

void main()
{
    for(int i=1,s=0;i<30;i++){
        if(i%2 == 0 || i%3 == 0) continue;
        cout << i << ' ';
        s += i;
    }
    cout << s << endl;
}

```

9. #include<iostream.h>

```

const int T=6;
void main()
{
    int i,j,k=0;
    for(i=1;i<=T;i+=2)
        for(j=2;j<=T;j++)
            if(i+j==T) cout << '+';
            else if(i*j==T) cout << '*';
            else k++;
    cout << endl << "k= " << k << endl;
}

```

10. #include<iostream.h>

```

void main()
{
    int a,b,c=0;
    for(a=1;a<6;a++){
        for(b=6;b>1;b--){
            if((a+b)%3==2) (c+=a+b;cout << a << ' ' << b << endl;)
            if(c>20) break;
        }
        cout << "c= " << c << endl;
    }
}

```

11. #include<iostream.h>

```

const int B=2;
void main()
{
    int p=1,s=1;
    while(s<50){
        p*=B;
        s+=p;
    }
    cout << "s= " << s << endl;
}

```

12. #include<iostream.h>

```

void main()
{

```



```

int x,y;
int i=2,p=1;
cout << "请输入两个正整数 x 和 y: ";
cin >> x >> y;
do {
    while(x%i == 0 && y%i == 0) {
        p *= i;
        x /= i;
        y /= i;
    }
    i++;
}while(x >= i && y >= i);
cout << "x 和 y 的最小公倍数: " << p * x * y << endl;
}

```

假定从键盘上输入的两个正整数为 24 和 88。

### (三) 指出下列每个程序的功能并上机验证

1. 

```

#include<iostream.h>
double fl(int n) {
    double sign=1,s=1;
    for(int i=2;i <=n; i++) {
        s += sign/(i * i);
        sign = -1;
    }
    return s;
}
void main()
{
    int a;
    cin >> a;
    cout << fl(a) << endl;
}

```
2. 

```

#include<iostream.h>
void main()
{
    double a,x,y;
    cin >> a;
    do {
        cin >> x;
        if(x == -1) break;
        if(x<0) y = a * x + 5;
        else if(x <= 20) y = 3 * x * x - 2 * a + 1;
        else y = a * a + x * x;
        cout << x << ' ' << y << endl;
    }while(1);
}

```
3. 

```

#include<iostream.h>

```

```

#include<stdlib.h>
double f1(double a, double b, char op) {
    switch(op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': if(b == 0) {
            cout << "divided by 0!" << endl;
            exit(1);
        }
        else return a/b;
        default: cout << "operator error!" << endl;
            exit(1);
    }
}
void main()
{
    double x,y;
    char op;
    cout << "请输入两个实数和一个运算符: ";
    cin >> x >> y >> op;
    cout << f1(x,y,op) << endl;
}

```

```

4. #include<iostream.h>
#include<math.h>
void main()
{
    int x1,x2;
    cin >> x1;
    x2 = int(sqrt(x1));
    for(int i = 1; i <= x2; i++)
        if(x1 % i == 0) cout << i << ' ' << x1/i << endl;
}

```

```

5. #include<iostream.h>
void main()
{
    int i,p=1,s=0;
    int N;
    cout << "输入一个正整数: ";
    cin >> N;
    for(i = 1; i <= N; i++) {
        p * = i;
        s += p;
    }
    cout << s << endl;
}

```

```

6. #include<iostream.h>
double f1(double x, int n)

```

```

{
    double p1,p2,p3,s;
    p1 = s = x; p2 = p3 = 1;
    if(r == 0) return s;
    for(int i = 1; i <= n; i++) {
        p1 *= x * x;
        p2 *= 2 * i * (2 * i + 1);
        p3 *= -1;
        s += p3 * p1 / p2;
    }
    return s;
}
void main()
{
    double x,y;
    int r;
    cout << "输入一个实数: ";
    cin >> x;
    cout << "输入一个正整数: ";
    cin >> n;
    y = f1(x,n);
    cout << "y = " << y << endl;
}

```

```

7. #include <iostream.h>
void main()
{
    int i,j;
    for(i=1;i <=5;i++) {
        for(j=1;j <=9;j++)
            if(j <= 5-i || j >= 5+i) cout << ' ';
            else cout << '*';
        cout << endl;
    }
}

```

```

8. #include <iostream.h>
int f1(int a, int b)
{
    int r;
    while(b != 0) {
        r = a % b;
        a = b; b = r;
    }
    return a;
}
int f2(int a, int b)
{
    int i = 2, p = 1;
    do {
        while(a % i == 0 && b % i == 0) {

```

```

        p *= i; a /= i; b /= i;
    }
    i++;
    }while(a >= i && b >= i);
    return p * a * b;
}
void main()
{
    int a,b;
    cout << "输入两个正整数: ";
    cin >> a >> b;
    cout << f1(a,b) << ' ' << f2(a,b) << endl;
}

```

```

9. #include <iostream.h>
   #include <stdlib.h>
   #include <time.h>
   const N=10;
   int ff(int x, int y) {
       int z;
       cout << x << '+' << y << '=';
       cin >> z;
       if(x+y == z) return 1; else return 0;
   }
   void main()
   {
       int a,b,c=0;
       srand(time(0));
       for(int i=0;i<N;i++) {
           a = rand() % 20 + 1;
           b = rand() % 20 + 1;
           c += ff(a,b);
       }
       cout << "得分: " << c * 10 << endl;
   }

```

```

10. #include <iostream.h>
    #include <stdlib.h>
    #include <math.h>
    void main()
    {
        double a,b,c;
        cout << "输入一元二次方程的二次项系数、一次项系数和常数项: " << endl;
        cin >> a >> b >> c;
        double d = b * b - 4 * a * c;
        if(d < 0.0) {
            cout << "此方程没有实根! " << endl;
            exit(1);
        }
        double x1,x2;
        if(d == 0.0)

```

```

        x1 = x2 = -b/(2*a);
    else {
        x1 = (-b + sqrt(d))/(2*a);
        x2 = (-b - sqrt(d))/(2*a);
    }
    cout << "此方程的两个根为:" << endl;
    cout << "x1 = " << x1 << endl;
    cout << "x2 = " << x2 << endl;
}

```

#### (四) 编写下列程序

1. 某城市为鼓励节约用水,对居民用水量作如下规定:若每人每月用水量不超过 2 吨,则按 0.3 元收费;若大于 2 吨但不超过 4 吨,则其中 2 吨按 0.3 元收费,剩余部分按每吨 0.6 元收费;若超过 4 吨,则其中 2 吨按 0.3 元收费,再有 2 吨按 0.6 元收费,剩余部分按每吨 1.2 元收费。试根据一户居民的月用水量和该户人口数计算出应交纳的水费。

2. 某班级学生进行百米跑测试,规定成绩在 12 秒以内(含 12 秒)为优秀,在 12 秒以上至 15 秒为达标,在 15 秒以上为不达标,编一程序,从键盘上输入每个人的成绩,分别统计出成绩为优秀、达标和不达标各多少人?各占学生总数的百分比是多少?

3. 计算  $1+3+3^2+\dots+3^{10}$  的值。

4. 求满足不等式  $2^2+4^2+\dots+n^2<1000$  的最大  $n$  值。

5.  $y = \begin{cases} \sqrt{a^2+x^2} & (x \leq 0) \\ 3a^3x^2+4ax-1 & (x > 0) \end{cases}$ , 求当  $x$  分别取 -3.8, 6.4, 2.3, -4.2, 8.9, 3.5, -5.0,

4.5 时所对应的  $y$  值,要求把  $a$  定义为常量,其初值由用户设定, $x$  的每个值由键盘输入。

6. 求出从键盘上输入的 10 个整数中的最大值。

7. 已知  $6 \leq a \leq 30, 15 \leq b \leq 36$ , 求满足不定方程  $2a+5b=126$  的全部整数解。如(13, 20)就是一个整数解。

8. 假定有 100 名中小學生参加义务植树活动,共植树 100 棵,其中高中生每人植 3 棵,初中生每人植两棵,小学生每两人植 1 棵。并且已知参加植树的高中、初中和小学生的人数均不小于 10 人,问他们各为多少人?

9. 已知  $y = 1 + \frac{1}{2}x + \frac{1}{3}x^2 + \dots + \frac{1}{10}x^9$ , 求  $x$  每取一个值时所对应的  $y$  值,其中  $x$  的每个值由键盘输入,直到输入终止标准 -100 为止。

10. 在输出窗口中显示出如下图形:

```

  * * * * *
 * * * * *
  * * * *
   * * *
    * *
     *

```

## 第四章 数组和字符串

### 4.1 数组的概念

在程序设计中存储单个数据时,需要根据数据的类型定义相应的变量来保存。如存储一个整数时需要定义一个整数变量来保存,存储一个实数时需要定义一个单精度或双精度变量来保存,存储含有多个成分的一个记录数据时,需要定义该类型的一个结构变量来保存。

若在程序设计中需要存储同一数据类型的、彼此相关的多个数据时,如存储数学上使用的一个数列或一个矩阵中的全部数据时,显然采用定义简单变量的方法是不行的,这就要求定义出能够同时存储多个值的变量,这种变量在程序设计中称为数组。

在实际应用中,一组相关的数据之间可能存在着一维关系,也可能存在着二维关系,等等。一个数列中的数据为一维关系,它除第一个数据外,每个数据只有一个直接前驱;除最后一个数据外,每个数据只有一个直接后继。假定一个数列为(38,42,25,60),则每个数的后一个数就是它的直接后继,每一个数的前一个数就是它的直接前驱,如42的直接前驱为38,直接后继为25。一个矩阵中的数据为二维关系,它除第一行和第一列上的所有数据外,每个数据在行和列的方向上各有一个直接前驱;除最后一行和最后一列上的所有数据外,每个数据在行和列的方向上各有一个直接后继。假定一个矩阵为:

$$\begin{bmatrix} 2 & 6 & 9 & 12 \\ 8 & 4 & 7 & 3 \\ 5 & 1 & 6 & 8 \end{bmatrix}$$

则每一个元素均处于相应行和列的交点位置上,虽然有的元素值相同,但由于所处的位置不同,所以是不同的元素。

在程序设计中,用一维数组表示和存储一维相关的数据;用二维数组表示和存储二维相关的数据,用三维数组表示和存储三维相关的数据,等等。假定一个数列为 $a_1, a_2, \dots, a_n$ ,则需要用一个一维数组来存储,假定仍用 $a$ 作为数组名,则 $a$ 中应至少包含有 $n$ 个元素,每个元素用来存储数列中一个相应的数据。若 $a$ 中正好包含有 $n$ 个元素,则这 $n$ 个元素依次表示为 $a[0], a[1], \dots, a[n-1]$ ,用 $a[0]$ 存储数列中的第一个数据 $a_1$ ,用 $a[1]$ 存储数列中的第二个数据 $a_2$ ,依次类推。假定一个矩阵为:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

则需要用一个二维数组来存储,假定二维数组名用 $b$ 表示,则 $b$ 中应至少包含 $m \times n$ 个元素,

也就是说,第一维尺寸至少为  $m$ ,第二维尺寸至少为  $n$ ,该数组  $b$  中的每个元素用来存储矩阵中的一个相应的数据。

## 4.2 数组的定义

### 4.2.1 一维数组

#### 1. 定义格式

一维数组同简单变量一样,也是通过变量定义语句定义的。其定义格式为:

<类型关键字> <数组名> [ $\langle$ 常量表达式 $\rangle$ ] [= ( $\langle$ 初值表 $\rangle$ )];

<类型关键字> 为已存在的一种数据类型,<数组名> 是用户定义的一个标识符,用它来表示一个数组,<常量表达式> 的值是一个整数,由它标明该数组的长度,即数组中所含元素的个数,每个元素具有<类型关键字> 所指定的类型,<常量表达式> 两边的中括号是语法所要求的符号,不是标明其内容为可选而使用的符号,<初值表> 是用逗号分开的一组表达式,每个表达式的值将被赋给数组中的相应元素。

当数组定义中包含有初始化选项时,其<常量表达式> 可以被省略,此时所定义的数组的长度将是<初值表> 中所含的表达式的个数。

一个数组被定义后,系统将在内存中为它分配一块含有  $n$  个( $n$  为数组长度)存储单元的存储空间,每个存储单元包含的字节数等于元素类型的长度。如对于一个含有 10 个 `int` 型元素的数组,它将对  $10 \times 4 = 40$  个字节的存储空间。

定义了一个数组,就相当于同时定义了它所含的每个元素。数组中的每个元素是通过下标运算符`[]`来指明和访问的,具体格式为:“<数组名>[<下标>]”,这与数组的定义格式相同,但出现的位置是不同的,当出现在变量定义语句时则为数组定义,而当出现在表达式中时则为一个元素。

对于一个含有  $n$  个元素的数组,C++ 语言规定:它的下标依次为  $0, 1, 2, \dots, n-1$ ,因此全部  $n$  个元素依次为  $a[0], a[1], a[2], \dots, a[n-1]$ ,其中假定  $a$  为数组名。

#### 2. 格式举例

- (1) `int a[20];`
- (2) `double b[MS];` // 假定 `MS` 为已定义的整型常量
- (3) `int c[5] = {1, 2, 3, 4, 0};`
- (4) `char d[] = {'a', 'b', 'c', 'd'};`
- (5) `int e[8] = {1, 4, 7};`
- (6) `char f[10] = {'B', 'A', 'S', 'I', 'C'};`
- (7) `bool g[2 * N + 1];` // 假定 `N` 为已定义的整型常量
- (8) `float h1[5], h2[10];`
- (9) `short x = 1, y = 2, z, w[4] = {25 + x, -10, x + 2 * y, 44};`

(10) `int p[];`

第一条语句定义了一个元素为 `int` 型、数组名为 `a`、包含 20 个元素的数组,所含元素依次为 `a[0], a[1], ..., a[19]`,每个元素同个 `int` 型简单变量一样,占用 4 个字节的存储空间,用来存储一个整数,整个数组占用 80 个字节的存储空间,用来存储 20 个整数。

第二条语句定义了一个元素类型为 `double`、数组长度为 `MS` 的数组 `b`,该数组占用 `MS × 8` 个字节的存储空间,能够用来存储 `MS` 个双精度数,数组 `b` 中的元素依次为 `b[0], b[1], ..., b[MS - 1]`。

第三条语句定义了一个整型数组 `c`,即元素类型为整型的数组 `c`,它的长度为 5,所含元素依次为 `c[0], c[1], c[2], c[3]` 和 `c[4]`,并相应被初始化为 1, 2, 3, 4 和 0。

第四条语句定义了一个字符数组 `d`,由于没有显式地给出它的长度,所以隐含为初值表中表达式的个数 4,该数组的 4 个元素 `d[0], d[1], d[2]` 和 `d[3]` 依次被初始化为 'a', 'b', 'c' 和 'd'。注意若没有给出数组的初始化选项,则表示数组长度的常量表达式不能省略。

第五条语句定义了一个含有 8 个元素的整型数组 `e`,它的初始化数据项的个数为 3,小于数组中元素的个数 8,这是允许的,这种情况的初始化过程为:将利用初始化表对前面相应元素进行初始化,而对后面剩余的元素则自动初始化为常数 0。数组 `e` 中的 8 个元素被初始化后得到的结果为: `e[0] = 1, e[1] = 4, e[2] = 7, e[3] ~ e[7] = 0`。

第六条语句定义了一个字符数组 `f`,它包含有 10 个字符元素,其中前 5 个元素被初始化为初值表所给的相应值,后 5 个元素被初始化为字符 '\0',对应数值为 0。

第七条语句定义了一个布尔型数组 `g`,它的数组长度为 `2 × N + 1`,每个元素没有被初始化。

第八条语句定义了两个单精度型一维数组 `h1` 和 `h2`,它们的数组长度分别为 5 和 10。在一条变量定义语句中,可以同时定义任意多个简单变量和数组,每两个相邻定义项之间必须用逗号分开。

第九条语句定义了三个短整型简单变量 `x, y` 和 `z`,其中 `x` 和 `y` 被初始化为 1 和 2,定义了一个短整型数组 `w`,它包含有四个元素,其中 `w[0]` 被初始化为 `25 + x` 的值,即 26, `w[1]` 被初始化为 -10, `w[2]` 被初始化为 `x + 2 * y` 的值,即 5, `w[3]` 被初始化为 44。

第十条语句是错误的数组定义,因为它既省略了数组长度选项,又省略了初始化选项,使系统无法确定该数组的大小,从而无法分配给它确定的存储空间。

### 3. 数组元素的访问

通过变量定义语句定义了一个数组后,用户便可以随时使用其中的任何元素。数组元素的使用是通过下标运算符 `[]` 指明和访问的,其中运算符左边为数组名,中间为下标。一个数组元素又称为下标变量,所使用的下标可以为常量,也可以为变量或表达式,但其值必须是整数,否则将产生编译错误。

假定 `a[n]` 为一个已定义的数组,则下面都是访问该数组的下标变量的合法格式:

- (1) `a[5]`                   // 下标为一个常数
- (2) `a[i]`                   // 下标为一个变量
- (3) `a[j++]`               // 下标为后增 1 表达式
- (4) `a[2 * x + 1]`       // 下标为一般表达式



假定在上述每个变量的下标表达式中,所使用的变量  $i, j$  和  $x$  的值分别为 2, 3 和 4, 则  $a[i]$  对应的数组元素为  $a[2]$ ,  $a[j++]$  对应的数组元素为  $a[3]$ , 同时  $j$  的值被修改为 4,  $a[2 * x + 1]$  对应的数组元素为  $a[9]$ 。

使用一个下标变量同使用一个简单变量一样,可以对它赋值,也可以取出它的值。如:

```
(1) int a[5] = {0, 1, 2, 3, 8};    // 定义数组 a 并进行初始化
(2) a[0] = 4;                    // 把 4 赋给 a[0]
(3) a[1] += a[0];                 // 把 a[0] 的值 4 累加到 a[1], 使 a[1] 的值变为 5
(4) a[3] = 3 * a[2] + 1;          // 把赋值号右边表达式的值 7 赋给 a[3]
(5) cout << a[a[0]];             // 因 a[0] = 4, 所以 a[a[0]] 对应的元素为 a[4],
                                   // 该语句输出 a[4] 的值 8
```

C++ 语言对数组元素的下标值不作任何检查,也就是说,当下标值超出它的有效变化范围  $0 \sim n-1$  (假定  $n$  为数组长度) 时,也不会给出任何出错信息。为了防止下标值越界(即小于 0 或大于  $n-1$ ),则需要编程者对下标值进行有效性检查。如:

```
(1) int a[5];
(2) for(int i=0; i<5; i++) a[i] = i * i;
(3) for(i=0; i<5; i++) cout << a[i] << ' ';
```

第一条语句定义了一个数组  $a$ , 其长度为 5, 下标变化范围为  $0 \sim 4$ 。第二条语句让循环变量  $i$  在数组  $a$  下标的有效范围内变化, 使下标为  $i$  的元素赋值为  $i$  的平方值, 该循环执行后数组元素  $a[0], a[1], a[2], a[3]$  和  $a[4]$  的值依次为 0, 1, 4, 9 和 16。第三条语句控制输出数组  $a$  中每一个元素的值, 输出语句中下标变量  $a[i]$  中的下标  $i$  的值不会超出它的有效范围。如果在第三条语句中, 用做循环判断条件的  $<$  表达式 2 不是  $i < 5$ , 而是  $i \leq 5$ , 则虽然  $a[5]$  不属于数组  $a$  的元素, 但也同样会输出它的值, 而从编程者角度来看是一种错误。由于 C++ 系统不对元素的下标值进行有效性检查, 所以用户必须通过程序检查, 确保其下标值有效。

#### 4. 程序举例

```
(1) #include <iostream.h>
    void main()
    {
        int i, a[6];
        for(i=0; i<6; i++) cin >> a[i];
        for(i=5; i>=0; i--) cout << a[i] << ' ';
        cout << endl;
    }
```

在程序的主函数中首先定义了一个  $\text{int}$  型简单变量  $i$  和一个含有 6 个  $\text{int}$  型元素的数组  $a$ , 接着使数组  $a$  中的每一个元素依次从键盘上得到一个相应的整数, 最后使数组  $a$  中的每一个元素的值按下标从大到小的次序显示出来, 每个值之后显示出一个空格, 以便使相邻的元素值分开。

若程序运行时, 从键盘上输入 3, 8, 12, 6, 20, 15 这 6 个整数, 则得到的输入和运行结果为:

```
3 8 12 6 20 15
15 20 6 12 8 3
```

```
(2) #include<iostream.h>
void main()
{
    int a[8] = {25,64,38,40,75,66,38,54};
    int max = a[0];
    for(int i = 1; i < 8; i++)
        if(a[i] > max) max = a[i];
    cout << "max: " << max << endl;
}
```

在这个程序的主函数中,第一条语句定义了一个整型数组 `a[8]`,并对它进行了初始化;第二条语句定义了一个整型变量 `max`,并用数组 `a` 中第一个元素 `a[0]` 的值初始化;第三条语句是一个 `for` 循环,它让循环变量 `i` 从 1 依次取值到 7,依次使数组 `a` 中的每一个元素 `a[i]` 同 `max` 进行比较,若元素值大于 `max` 的值,则就把它赋给 `max`,使 `max` 始终保存着从 `a[0] ~ a[i]` 元素之间的最大值,当循环结束后,`max` 的值就是数组 `a` 中所有元素的最大值;第四条语句输出 `max` 的值。

在该程序的执行过程中,`max` 依次取 `a[0]`,`a[1]` 和 `a[4]` 的值,不会取其他元素的值。程序运行结果为:

```
max:75
```

```
(3) #include<iostream.h>
const int N = 7;
void main()
{
    double w[N] = {2.6,7.3,4.2,5.4,6.2,3.8,1.4};
    int i,x;
    cout << "输入一个实数: ";
    cin >> x;
    for(i = 0; i < N; i++)
        if(w[i] > x) cout << "w[" << i << "] = " << w[i] << endl;
}
```

此程序的功能是从数组 `a[N]` 中顺序查找出比 `x` 值大的所有元素并显示出来。若从键盘上输入的 `x` 值为 5.0,则得到的程序运行结果为:

```
输入一个实数:5.0
w[1] = 7.3
w[3] = 5.4
w[4] = 6.2
```

```
(4) #include<iostream.h>
const int M = 10;
void main()
{
    int a[M+1];
    a[0] = 1; a[1] = 2;
```

```

int i;
for(i = 2; i <= M; ++i)
    a[i] = a[i-1] + a[i-2];
for(i = 0; i < M; ++i)
    cout << a[i] << ' ';
cout << a[M] << endl;
}

```

该程序首先定义数组 `a`, 并分别为数组元素 `a[0]` 和 `a[1]` 赋值 1 和 2, 接着依次计算出 `a[2]` 至 `a[M]` 的值, 每个元素值均等于它的前两个元素值之和, 最后按照下标从小到大的次序显示出数组 `a` 中每个元素的值。该程序运行结果为:

1,2,3,5,8,13,21,34,55,89,144

## 4.2.2 二维数组

### 1. 定义格式

二维数组同一维数组一样, 也是通过变量定义语句定义的, 其定义格式为:

```

<类型关键字> <数组名> [<常量表达式 1>] [<常量表达式 2>]
    [= {(<初值表 1>), (<初值表 2>), ...}];

```

在上述定义格式中, `<常量表达式 1>` 和 `<常量表达式 2>` 两边的中括号也同一维数组定义中 `<常量表达式>` 两边的中括号的用法相同, 都是语法所要求的符号, 不是指一般规定的其内容为任选项的标识。

二维数组定义中的 `<常量表达式 1>` 和 `<常量表达式 2>` 分别指定数组的第一维下标 (又称为行下标) 和第二维下标 (又称为列下标) 取值的个数, 假定 `<常量表达式 1>` 和 `<常量表达式 2>` 的值分别为 `m` 和 `n`, 则行下标的取值范围是 `0~m-1` 之间的 `m` 个整数, 列下标的取值范围是 `0~n-1` 之间的 `n` 个整数。

对于一个行下标取值个数为 `m`, 列下标取值个数为 `n` 的二维数组 `a`, 它所含元素的个数为 `m×n`, 即数组长度为 `m×n`, 每一个元素含有两个下标, 具体表示为: “`<数组名>[<行下标>][<列下标>]`”, 数组 `a` 中的所有元素表示为:

```

a[0][0]    a[0][1]    ...    a[0][n-1]
a[1][0]    a[1][1]    ...    a[1][n-1]
⋮          ⋮          ⋮          ⋮
a[m-1][0]  a[m-1][1]  ...    a[m-1][n-1]

```

我们知道, 当定义了一个一维数组后, 系统为它分配一块连续的存储空间, 该空间的大小为 `n×sizeof(<元素类型>)`, 其中 `n` 为一维数组长度。

在 C++ 系统中, 数组名同时表示该数组占用的存储空间的首地址。例如, 若定义了一个 `int` 型的一维数组 `b[10]`, 则下标为 `i` 的元素 `b[i]` 的地址为 `b+4*i`, 其中 `0≤i≤9`。在内存中数组 `b` 的存储分配示意图为:

0	1	2	3	4	5	6	7	8	9
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]
0	4	8	12	16	20	24	28	32	36

其中每个方框表示一个元素的存储单元,它的上面为该元素的下标,也是存储单元的顺序编号,下面为该元素相对于首地址 b 的偏移地址。

当定义了一个二维数组后,系统也同样为它分配一块连续的存储空间,该存储空间的大小为  $m \times n \times \text{sizeof}(\text{元素类型})$ ,其中 m 和 n 分别表示第一维下标和第二维下标的取值个数。

系统给一个二维数组中的所有元素分配存储单元时,是首先按行下标从小到大的次序,行下标相同再按列下标从小到大的次序进行的。例如,若定义了一个 double 型的二维数组 c[M][N],则任一元素 c[i][j] 的地址为  $c + (i \times N + j) \times 8$ ,其中  $0 \leq i \leq M - 1$ ,  $0 \leq j \leq N - 1$ 。假定常量 M 和 N 分别为 4 和 2,则数组 c 的存储分配示意图为:

0	1	2	3	4	5	6	7
c[0][0]	c[0][1]	c[1][0]	c[1][1]	c[2][0]	c[2][1]	c[3][0]	c[3][1]
0	8	16	24	32	40	48	56

同一维数组的存储分配示意图一样,每个方框表示一个元素的存储单元,它的上面为存储单元的顺序编号,下面为该元素相对于首地址 c 的偏移地址。

若要计算 c[2][1] 的存储地址,则为  $c + (2 \times 2 + 1) \times 8 = c + 40$ 。

若在二维数组的定义格式中,包含有最后的初始化选项,则能够在定义二维数组的同时,对所有元素进行初始化,其中每个用花括号括起来的初值表用于初始化数组中的一行元素,即 <初值表 1> 用于初始化行下标为 0 的所有元素,<初值表 2> 用于初始化行下标为 1 的所有元素,依次类推。同一维数组的初始化一样,若有的元素没有对应的初始化数据,则自动对它初始化为 0。

在二维数组的定义格式中,若带有初始化选项,则 <常量表达式 1> 可以省略,此时将定义一个行数等于初值表个数的二维数组。

## 2. 格式举例

- (1) `int a[3][3];`
- (2) `double b[M][N];` // 假定 M 和 N 为整型常量
- (3) `int c[2][4] = {{1,3,5,7},{2,4,6,8}};`
- (4) `int d[][3] = {{0,1,2},{3,4,5},{6,7,8}};`
- (5) `int e[3][4] = {{0},{1,2}};`
- (6) `char f[CN+1][CN+1], c1='a', c2;` // 假定 CN 为整型常量
- (7) `int g[10], h[10][5];`
- (8) `int r[][5];`

第一条语句定义了一个二维数组 a[3][3],它包含有 9 个元素,元素类型为 int,每个元素同一个 int 型简单变量一样,能够用来表示和存储一个整数。

第二条语句定义了一个元素类型为 `double` 的二维数组 `b[M][N]`, 它包含  $M \times N$  个元素, 每个元素用来保存一个实数, 元素中行下标的有效范围为  $0 \sim M-1$ , 列下标的有效范围为  $0 \sim N-1$ , 任一元素 `b[i][j]` 的存储地址为  $b + (i \times N + j) \times 8$ , 当然  $i$  和  $j$  都要在有效取值范围以内。

第三条语句定义了一个元素类型为 `int` 的二维数组 `c[2][4]`, 并对该数组进行了初始化, 使得 `c[0][0]`, `c[0][1]`, `c[0][2]` 和 `c[0][3]` 的初值分别为 1, 3, 5 和 7; `c[1][0]`, `c[1][1]`, `c[1][2]` 和 `c[1][3]` 的初值分别为 2, 4, 6 和 8。

第四条语句定义了一个元素类型为 `int` 的二维数组 `d`, 它的列下标的取值范围为  $0 \sim 2$ , 行下标的取值范围没有显式给出, 但由于给出了初始化选项, 并且含有三个初值表, 所以取值范围隐含为  $0 \sim 2$ , 相当于在数组定义的第一个中括号内省略了行下标取值个数 3。

第五条语句定义了一个元素类型为 `int` 的二维数组 `e[3][4]`, 它的第 1 行 (即行下标为 0) 的四个元素被初始化为 0, 第 2 行的四个元素 `e[1][0]`, `e[1][1]`, `e[1][2]` 和 `e[1][3]` 分别被初始化为 1, 2, 0 和 0, 第 3 行的四个元素也均被初始化为 0。

第六条语句定义了一个元素类型为 `char` 的二维数组 `f`, 它的行、列下标的上界均为 `CN`, 其取值均为  $0 \sim \text{CN}$  之间的整数, 该语句同时定义了字符变量 `c1` 和 `c2`, 并使 `c1` 初始化为字符 'a'。

第七条语句同时定义了两个元素类型为 `int` 的数组, 一个为一维数组 `g[10]`, 另一个为二维数组 `h[10][5]`, 它们分别含有 10 个元素和 50 个元素, 每个元素能够表示和存储一个整数。

第八条语句定义的二维数组 `r` 是错误的, 因为它既没有给出第一维下标的取值个数, 又没有给出初始化选项, 所以系统无法确定该数组的长度, 从而无法为它分配一定大小的存储空间。

### 3. 数组元素的访问

一个二维数组被定义后, 与使用一维数组一样, 是通过下标运算符指明和访问元素, 其中对行下标和列下标都要进行运算才能够惟一指定一个元素。二维数组中的一个元素由于使用了两个下标, 所以又称为双下标变量。一个双下标变量中的任一个下标不仅可以为常量, 同样可以为变量或表达式, 当然它们都必须为整数类型。如:

- (1) `a[2][3]`           // 每个下标均为常量
- (2) `a[i][j]`           // 每个下标均为变量
- (3) `a[i][5]`           // 行下标为变量, 列下标为常数
- (4) `a[i-1][j+1]`   // 每个下标均为表达式

若  $i$  和  $j$  的值分别为 2 和 3, 则上述下标变量 `a[i][j]` 对应的元素为 `a[2][3]`, `a[i][5]` 对应的元素为 `a[2][5]`, `a[i-1][j+1]` 对应的元素为 `a[1][4]`。

使用双下标变量同使用单下标变量和简单变量一样, 既可以用它存储数据, 又可以取出它的值参加运算。如:

- (1) `int a[4][5];`                   // 定义数组
- (2) `a[1][2] = 6;`                   // 向 `a[1][2]` 元素赋值 6
- (3) `a[2][2] = 3 * a[1][2] + 1;`   // 取出 `a[1][2]` 的值 6 参与运算,

```

// 把赋值号右边表达式的值 19 赋给 a[2][2]元素中
(4) a[i][j-1] = a[i][j]; // 把 a[i][j]的值赋给 a[i][j-1]元素中
(5) cout << a[1][2] * a[2][2] - 3 << endl; // 输出表达式的值 111 到显示窗口上

```

C++ 系统对待二维下标变量同样不作下标有效性检查,所以也需要编程者通过程序进行检查处理,避免下标越界的情况发生。

在C++ 语言中,不仅可以定义和使用一维数组和二维数组,也可以定义和使用三维及更高维的数组。如,下面的语句定义了一个三维数组:

```
int s[P][M][N]; // 假定 P,M,N 均为已定义的整型常量
```

该数组的数组名为 s,第一维下标的取值范围为 0~P-1,第二维下标的取值范围为 0~M-1,第三维下标的取值范围为 0~N-1。该数组共包含 P×M×N 个 int 型的元素,共占用 P×M×N×4 个字节的存储空间。数组中的每个元素由三个下标惟一确定,如 s[1][0][3]就是该数组中的一个元素(假定 P,M 和 N 分别大于等于 2,1 和 4)。

若用一个三维数组来表示一本书,则第一维表示页,第二维表示页内的行,第三维表示行内一个字符位置所在的列,数组中每个元素的值就是相应位置上的字符。

#### 4. 程序举例

```

(1) #include <iomanip.h>
    const int M=3,N=4;
    void main()
    {
        int a[M][N] = {{7,5,14,3},{6,20,7,8},{14,6,9,18}};
        int i,j;
        for(i=0;i<M;i++){
            for(j=0;j<N;j++){
                cout << setw(5) << a[i][j];
            }
            cout << endl;
        }
    }

```

该程序首先定义了一个元素为 int 类型的二维数组 a[M][N],并对它进行了初始化;接着通过双重 for 循环输出每一个元素的值,其中外循环变量 i 控制行下标从小到大依次变化,内循环变量 j 控制列下标从小到大依次变化,每输出一个元素值占用显示窗口的 5 个字符宽度,当同一行元素(即行下标值相同的元素)输出完毕后,将输出一个换行符,以便下一行元素从显示窗口的下一行显示出来。该程序的运行结果为:

```

7      5      14      3
6      20      7      8
14      6      9      18

```

```

(2) #include <iostream.h>
    void main()
    {
        int b[2][5] = {{7,15,2,8,20},{12,25,37,16,28}};
        int i,j,k=b[0][0];
        for(i=0;i<2;i++){

```

```

        for(j = 0; j < 5; j++)
            if(b[i][j] > k) k = b[i][j];
    cout << k << endl;
}

```

在这个程序中首先定义了元素类型为 `int` 的二维数组 `b[2][5]` 并初始化,接着定义了 `int` 型的简单变量 `i, j, k`, 并对 `k` 初始化为 `b[0][0]` 的值 7, 然后使用双重 `for` 循环依次访问数组 `b` 中的每个元素, 并且每次把大于 `k` 的元素值赋给 `k`, 循环结束后 `k` 中将保存着所有元素的最大值, 并被输出出来, 这个值就是 `b[1][2]` 的值 37。

```

(3) #include <iostream.h>
    const int M = 4;
    void main()
    {
        int c[M] = {0};
        int d[M][3] = {{1, 5, 7}, {3, 2, 10}, {6, 7, 9}, {4, 3, 7}};
        int i, j, sum = 0;
        for(i = 0; i < M; i++) {
            for(j = 0; j < 3; j++) c[i] += d[i][j];
            sum += c[i];
        }
        for(i = 0; i < M; i++) cout << c[i] << ' ';
        cout << sum << endl;
    }

```

该程序主函数中的第一条语句定义了一个一维数组 `c[M]` 并使每个元素初始化为 0, 第二条语句定义了一个二维数组 `d[M][3]` 并使每个元素按所给的数值初始化, 第三条语句定义了 `i, j` 和 `sum`, 并使 `sum` 初始化为 0, 第四条语句是一个双重 `for` 循环, 它依次访问数组 `d` 中的每个元素, 并把每个元素的值累加到数组 `c` 中与该元素的行下标值相同的对应元素中, 然后再把数组 `c` 中的这个元素值累加到 `sum` 变量中, 第五条语句依次输出数组 `c` 中的每个元素值, 第六条语句输出 `sum` 的值。该程序把二维数组 `d` 中的同一行元素值累加到一维数组 `c` 中的相应元素中, 把所有元素的值累加到简单变量 `sum` 中。该程序的运行结果为:

```
13 15 22 14 64
```

### 4.2.3 使用 `typedef` 语句定义数组类型

#### 1. 一维数组类型的定义格式

```
typedef <元素类型关键字> <数组类型名> [<常量表达式>];
```

例如:

- (1) `typedef int vector[10];`
- (2) `typedef char strings[80];`
- (3) `typedef short int array[N];`

第一条语句定义了一个元素类型为 `int`, 含有 10 个元素的数组类型 `vector`, 若不使用 `typedef` 保留字, 则就变成了数组定义, 它只定义了一个元素类型为 `int`、含有 10 个元素的数

组 `vector`。这两种定义有着本质的区别,若定义的是数组 `vector`,系统将为其分配有保存 10 个整数的存储单元,共 40 个字节的存储空间;若定义的是数组类型 `vector`,系统只是把该类型的有关信息登记下来,待以后利用该类型定义对象时使用,具体地说,就是把 `vector` 的元素类型 `int`,数组长度 10,数组类型名 `vector` 等登记下来,待以后定义 `vector` 类型的对象时使用。

第二条语句定义了一个元素类型为 `char`,含有 80 个元素的数组类型 `strings`,以后可以直接使用 `strings` 类型定义数组对象,每个数组对象的元素为 `char` 型,数组长度(即元素个数)为 80。

第三条语句定义了一个元素类型为 `short int` 的含有 `N` 个元素(`N` 为已定义的符号常量)的数组类型 `array`,以后利用它可以直接定义该类型的对象,它是一个含有 `N` 个短整型元素的数组。

下面是利用上述类型定义对象的一些例子。

(1) `vector v1, v2;`

(2) `strings s1, s2, s3 = "define type";`

(3) `array a = {25, 36, 19, 48, 44, 50};` // 假定常量 `N ≥ 6`

第一条语句定义了 `vector` 类型的两个对象 `v1` 和 `v2`,每个对象都是 `vector` 类型的一个数组,每个数组由 10 个整型元素所组成。

第二条语句定义了 `strings` 类型的三个对象 `s1, s2` 和 `s3`,并且对 `s3` 进行了初始化,每个对象都是含有 80 个字符空间的数组。

第三条语句定义了一个 `array` 类型的对象 `a`,它是一个含有 `N` 个短整型元素的数组,该语句同时对数组 `a` 进行了初始化,使得 `a[0] ~ a[5]` 的元素值依次为 25, 36, 19, 48, 44 和 50。

## 2. 二维数组类型的定义格式

`typedef <元素类型关键字> <数组类型名> [ <常量表达式 1> ] [ <常量表达式 2> ];`

例如:

(1) `typedef int matrix[5][5];`

(2) `typedef char nameTable[10][NN];`

(3) `typedef double DD[M+1][N+1];`

第一条语句定义了含有 5 行 5 列共 25 个 `int` 型元素的数组类型 `matrix`,第二条语句定义了 10 行 `NN` 列共  $10 \times NN$  个 `char` 型元素的数组类型 `nameTable`,第三条语句定义了含有 `M+1` 行 `N+1` 列共  $(M+1)(N+1)$  个 `double` 类型元素的数组类型 `DD`。

利用这三个二维数组类型可以直接定义出相应的二维数组。如:

(1) `matrix mx = {{0}};`

(2) `nameTable nt = { " " };` // 或使用等同的 `{{'\0'}}` 初始化

(3) `DD dd = {{0.0}};`

第一条语句定义了二维整型数组类型 `matrix` 的一个对象 `mx`,该对象是一个  $5 \times 5$  的二维整型数组,每个元素均被初始化为 0;第二条语句定义了二维字符数组类型 `nameTable` 的一个二维字符数组 `nt`,该数组中的每个元素均被初始化为空字符;第三条语句定义了二维双精度数组类型 `DD` 的一个数组 `dd`,它的每个元素均被初始化为 0.0。



在 typedef 语句中, <元素类型关键字> 可以是 C++ 语言中预定义的任何一种数据类型, 也可以是用户在前面已定义的任何一种数据类型, 所以通过该语句定义的类型同样可以用在其后的 typedef 语句中。如:

- (1) typedef vector vectorSet[20];
- (2) vectorSet vs;

第一条语句定义了元素类型为 vector, 元素个数为 20 的一个数组类型 vectorSet, 第二条语句定义了数据类型为 vectorSet 的一个对象 vs, 该对象包含有 20 个类型为 vector 的元素, 每个元素又包含有 10 个 int 类型的元素, 所以整个数组共包含有 20 行 10 列共 200 个整数元素, 它等同于对 vs 的如下定义:

```
int vs[20][10];
```

利用 typedef 语句同样可以定义更高维的数组类型, 这里就不进行讨论了。

### 3. 对已有类型定义别名

利用 typedef 语句不仅能够定义数组类型, 而且能够对已有类型定义出另一个类型名, 以此作为原类型的一个别名。如:

- (1) typedef int inData;
- (2) typedef char chData;
- (3) typedef char \* chPointer;

第一条语句对 int 类型定义了一个别名 inData, 第二条语句对 char 类型定义了一个别名 chData, 第三条语句对 char \* 类型(它是字符指针类型)定义了一个别名 chPointer。以后使用 inData, chData 和 chPointer 就如同分别使用 int, char 和 char \* 一样, 定义出相应的对象。如:

- (1) inData x, y;
- (2) inData a[5] = {1, 2, 3, 4, 5};
- (3) chData b1, b2 = 'a';
- (4) chData c[10] = "char data";
- (5) chPointer p = 0;

第一条语句定义了 inData(即 int)型的两个变量 x 和 y, 第二条语句定义了元素类型为 int 的一维数组 a[5]并进行了初始化, 第三条语句定义了 chData(即 char)型的两个变量 b1 和 b2, 并把 b2 初始化为 'a', 第四条语句定义了一个字符数组 c[10]并初始化为 "char data", 第五条语句定义了一个字符指针变量 p, 并初始化为 0(即 NULL)。

## 4.3 数组的应用

数组是表示和存储数据的一种重要方法, 利用数组能够进行计算、统计、排序、查找等各种运算。下面通过程序设计的例子来说明这些运算。

### 4.3.1 数值计算

例 1. 国家对个人月收入征收个人所得税的办法如表 4-1 所示, 编一程序, 根据一个人的月收入计算出应缴纳的税额和税后所得的金额。

表 4-1 个人月收入所得税表

级数	级距	税率(%)
1	800 元以下部分	0
2	800 ~ 1500 元之间部分	5
3	1500 ~ 3000 元之间部分	10
4	3000 ~ 6000 元之间部分	20
5	6000 ~ 9000 元之间部分	30
6	9000 ~ 12000 元之间部分	40
7	12000 元以上部分	45

分析: 由每一级的级距上界组成一个数列(最后一级的上界理论上为无穷大, 但计算机无法表示一个无穷大的数, 所以可用一个非常大的数, 如  $1e9$  来表示), 假定该数列用  $a$  表示; 由每一级税率组成另一个数列, 假定该数列用  $b$  表示, 则  $a$  和  $b$  分别为:

$$a = (800, 1500, 3000, 6000, 9000, 12000, 1e9)$$

$$b = (0, 0.05, 0.10, 0.20, 0.30, 0.40, 0.45)$$

设用  $x$  表示一个人的月收入, 用  $i$  表示  $x$  所对应的级数, 用  $y$  表示月收入为  $x$  应缴纳的税额, 则  $y$  的计算公式为:

$$y = (x - a_{i-1})b_i + \sum_{j=1}^{i-1} (a_j - a_{j-1})b_j$$

其中  $1 \leq i \leq 7$ ,  $a_1 \sim a_7$  依次为数列  $a$  中对应的级距上界,  $b_1 \sim b_7$  依次为数列  $b$  中对应的税率。如当  $x = 4500$  时, 对应的级距为 4, 应缴纳税额为:

$$\begin{aligned} y &= (x - a_3)b_4 + \sum_{j=1}^3 (a_j - a_{j-1})b_j \\ &= (4500 - 3000) \times 0.20 + (a_3 - a_2) \times b_3 + (a_2 - a_1) \times b_2 \\ &= 300 + (3000 - 1500) \times 0.10 + (1500 - 800) \times 0.05 \\ &= 300 + 150 + 35 \\ &= 485 \end{aligned}$$

在编写此题的程序时, 应首先说明存储数列  $a$  和  $b$  的两个一维数组, 假定仍用标识符  $a$  和  $b$  表示, 它们的长度应均为 8, 其中用  $a[i]$  和  $b[i]$  分别存储  $a_i$  和  $b_i$ , 下标为 0 的元素未用; 接着给  $x$  输入一个值, 并求出它对应的级数  $i$ ; 最后计算出  $y$  的值, 并打印出  $y$  和  $x - y$  的值, 它们分别为上缴税额和税后所得的金额。

根据分析, 编写出程序如下:

```
#include <iostream.h>
const int N=8;
void main()
```

```

{
    double a[N] = {0,800,1500,3000,6000,9000,12000,1e9};
    double b[N] = {0,0,0.05,0.10,0.20,0.30,0.40,0.45};
    double x,y;
    cout << "输入一个人的月收入(单位\ "元\ "):";
    cin >> x;
    int i,j;
    for(i=1;i<N;i++) if(x <= a[i]) break;
    y = (x - a[i-1]) * b[i];
    for(j=i-1;j >= 2;j--) y += (a[j] - a[j-1]) * b[j];
    cout << "所得税:" << y << endl;
    cout << "税后额:" << x - y << endl;
}

```

假定程序运行时,从键盘上输入 8000 作为 x 的值,则得到的运行结果为:

```

输入一个人的月收入(单位"元"):8000
所得税:1385
税后额:6615

```

例 2. 已知两个矩阵 A 和 B 如下,编一程序计算出它们的和。

$$A = \begin{bmatrix} 7 & -5 & 3 \\ 2 & 8 & -6 \\ 1 & -4 & -2 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 6 & -9 \\ 2 & -8 & 3 \\ 5 & -2 & -7 \end{bmatrix}$$

分析:由数学知识可知,行数和列数分别对应相同的两个矩阵可以做加法,它们的和仍为一个矩阵,并且与两个加数矩阵具有相同的行数和列数。此题中的两个矩阵均为 3 行 × 3 列,所以它们的和矩阵同样为 3 行 × 3 列。两矩阵加法运算的规则是,和矩阵中每个元素的值等于两个加数矩阵中对应位置上的元素值之和,即  $C_{ij} = A_{ij} + B_{ij}$ ,其中 A 和 B 表示两个加数矩阵,C 表示它们的和,即和矩阵。在程序中,首先应定义三个二维数组,假定分别用标识符 a,b 和 c 表示,分别对应 A,B 和 C 这三个矩阵,并需要对 a 和 b 进行初始化;接着根据 a 和 b 计算出 c;然后按照矩阵的书写格式输出数组 c,它就是对应的矩阵 C。

根据分析编写出程序如下:

```

#include <iomanip.h>
const int N=3;
void main()
{
    int a[N][N] = {{7, -5, 3}, {2, 8, -6}, {1, -4, -2}};
    int b[N][N] = {{3, 6, -9}, {2, -8, 3}, {5, -2, -7}};
    int i,j,c[N][N];
    for(i=0;i<N;i++) // 计算矩阵 C
        for(j=0;j<N;j++)
            c[i][j] = a[i][j] + b[i][j];
    for(i=0;i<N;i++) { // 输出矩阵 C
        for(j=0;j<N;j++)
            cout << setw(5) << c[i][j];
        cout << endl;
    }
}

```

该程序运行时得到的输出结果如下：

```
10    1    -6
 4     0    -3
 6    -6    -9
```

例3. 有一家公司,生产五种型号的产品,上半年各月份的产量如表4-2所示,每种型号产品的单价如表4-3所示,编一程序计算出该公司上半年的总产值。

表4-2 产量统计表

月份 \ 型号	TV-14	TV-18	TV-21	TV-25	TV-29
一	438	269	738	624	513
二	340	420	572	726	612
三	455	286	615	530	728
四	385	324	713	594	544
五	402	382	550	633	654
六	424	400	625	578	615

表4-3 单价表

型号	单价(元)
TV-14	500
TV-18	950
TV-21	1340
TV-25	2270
TV-29	2985

分析:表4-2需要用一个二维数组来存储,该数组的行下标表示月份,即用0~5依次表示1~6月份,该数组的列下标表示产品型号,即用0~4依次表示TV-14,TV-18,TV-21,TV-25和TV-29,数组中的每一元素值为相应月份和型号的产量。表4-3也需用一个一维数组来存储,该数组的下标依次对应每一种产品型号,每一元素值为该型号的单价。假定用b和c分别表示这两个数组,则此程序开始应定义它们并进行初始化。

要计算出上半年的总产值,首先必须计算出每月份的产值,然后再逐月累加起来。为此,设一维数组d[6]用来存储各月份的产值,即用d[0]存储一月份的产值,d[1]存储二月份的产值,依次类推。设用变量sum累加每一月份的产值,当从1月份累加到6月份之后,sum的值就是该公司上半年的总产值。根据数组b和c计算出第i+1月份产值的公式为:

$$d[i] = \sum_{j=0}^4 b[i][j] \times c[j] \quad (0 \leq i \leq 5)$$

根据分析,编写出此题的完整程序如下:

```
#include <iostream.h>
void main()
{
    int b[6][5] = {{438,269,738,624,513},{340,420,572,726,612},
                  {455,286,615,530,728},{385,324,713,594,544},
                  {402,382,550,633,654},{424,400,625,578,615}};
    int c[5] = {500,950,1340,2270,2985};
    int d[6] = {0};
    int sum=0;
    int i,j;
    for(i=0;i<6;i++){
        for(j=0;j<5;j++){ // 计算出第 i+1 月份的产值
            d[i] += b[i][j] * c[j];
        }
    }
    sum = d[0] + d[1] + d[2] + d[3] + d[4] + d[5];
    cout << "上半年总产值为: " << sum << endl;
}
```

```

        cout << d[i] << ' '; // 输出第 i+1 月份的产值
        sum += d[i];          // 把第 i+1 月份的产值累加到 sum 中
    }
    cout << endl << "sum: " << sum << endl; // 输出上半年总产值
}

```

若上机输入和运行该程序,则得到的输出结果为:

```

4411255 4810320 4699480 4427940 4690000 4577335
sum:27616330

```

### 4.3.2 统计

例 1. 有一个协会在换届选举中由全体会员无记名投票直选主席的活动,共有 5 名候选人,每个人的代号分别用 1,2,3,4,5 表示,每名会员填写一张选票,若同意某名候选人则在其姓名后打上对号即可,编一程序根据所有选票统计出每位候选人所得票数,其中每张选票上所投候选人的代号由键盘输入,当输入完所有选票后用 -1 作为终止数据输入的标志。

分析:由于需要分别统计 5 位候选人的票数,所以要同时使用 5 个统计变量,为此定义一个具有 6 个元素的一维整型数组,其中用下标为 1 的元素统计代号为 1 的候选人票数,用下标为 2 的元素统计代号为 2 的候选人票数,依次类推,而下标为 0 的元素未用。假定该数组为 a[6],则当从键盘上输入的一个代号为 1 时,就在元素 a[1]上加 1,为 2 时就在元素 a[2]上加 1,总之当输入的代号为 i( $1 \leq i \leq 5$ )时就在 a[i]上加 1。当统计结束后每个数组元素 a[i]的值就是代号为 i 的候选人最后所得的票数。

根据分析,编写出程序如下:

```

#include <iostream.h>
void main()
{
    int i, a[6] = {0};
    cout << "请依次输入每张选票上所投候选人的代号: ";
    cin >> i;
    while(i != -1) {
        if(i >= 1 && i <= 5) a[i]++;
        cin >> i;
    }
    cout << endl;
    for(i = 1; i <= 5; i++) cout << i << ': ' << a[i] << endl;
}

```

下面是程序一次运行的结果:

```

请依次输入每张选票上所投候选人的代号:1 2 3 2 4 3 5 1 3 4 2 3 5 -1

1:2
2:3
3:4
4:2
5:2

```

例2. 某社区对所属的  $N$  户居民进行用电量统计,每隔 50 度用电量为一个统计区间,但当大于等于 500 度时为一个统计区间,编一程序,分别统计每个用电区间内的居民户数。

分析:由题意可知,用电区间共有 11 个,其中 0~49 为第一个区间,50~99 为第二个区间,依次类推。为此定义一个统计数组,假定用  $c[11]$  表示,用它的第一个元素  $c[0]$  统计用电量为 0~49 区间内的用户数,用它的第二个元素  $c[1]$  统计用电量为 50~99 区间内的用户数,……,用它的第 11 个元素  $c[10]$  统计用电量为大于等于 500 度的用户数。

在程序的主函数中,应首先定义数组  $c[11]$  并初始化每个元素的值为 0;接着通过  $N$  次循环,从键盘上依次输入每户的用电量  $x$ ,并统计到相应的元素中,即下标为  $x/50$  的元素中,当然若  $x \geq 500$ ,则统计到  $c[10]$  元素中;最后通过循环输出在数组  $c$  中保存的统计结果。

根据分析编写出程序如下:

```
#include <iostream.h>
const int N=100; // 假定 N 的值为 100
void main()
{
    int c[11]={0};
    int i,x;
    for(i=1;i <= N;i++){
        cin >> x;
        if(x<500) c[x/50]++; else c[10]++;
    }
    for(i=0;i <= 10;i++) cout << "c[" << i << "] = " << c[i] << endl;
}
```

### 4.3.3 排序

例1. 已知有 10 个常数为 42,65,80,74,36,44,28,65,94,72,编一程序,采用选择排序方法,按照从小到大的顺序打印输出。

分析:首先需要把已知的 10 个常数存入到一维数组中,假定该数组被定义为  $a[10]$ ;接着采用选择排序的方法对数组  $a[10]$  中的 10 个元素按照其值从小到大的顺序排序,使得元素值的排列次序与下标次序相同,即得到  $a[0] \leq a[1] \leq a[2] \leq \dots \leq a[9]$ ;最后按照下标次序显示出每个元素的值,它们是按值的从小到大的次序排列的。

对数组  $a$  中的  $n$  个元素进行选择排序共需要进行  $n-1$  次选择和交换的过程,第一次从待排序区间  $a[0] \sim a[n-1]$  中通过顺序比较选择一个最小值元素,把它与该区间的第一个元素  $a[0]$  交换后, $a[0]$  就成为所有  $n$  个元素中的最小值;第二次从新的待排序区间  $a[1] \sim a[n-1]$  中通过顺序比较选择一个最小值元素,把它与当前区间的第一个元素  $a[1]$  交换后, $a[1]$  就成为仅次于  $a[0]$  的最小值元素;依次类推,第  $n-1$  次(即最后一次)从当前待排序区间  $a[n-2] \sim a[n-1]$  中通过顺序比较选择一个最小值元素,把它与当前区间的第一个元素  $a[n-2]$  交换后,整个排序过程结束,此时数组  $a$  中的所有  $n$  个元素就按照其值从小到大的次序排列了。

若一个数组中的元素是按照其值从小到大的次序排列的,则称之为有序表,否则称之为无序表。对于一个有序表若按照从小到大有序则又称为升序表或正序表,若按照从大到小

有序则又称为降序表或逆序表。通常若不特别指明,所说的有序均为升序。

选择排序过程使用双重 for 循环来实现,设外循环变量为  $i$ ,它需要从 1 顺序取值到  $n-1$ ,其中  $n$  为待排序数组中元素的个数,每次的待排序区间为  $a[i-1] \sim a[n-1]$ ,这里假定  $a$  为数组名,设内循环变量为  $j$ ,它需要从  $i$  顺序取值到  $n-1$ ,每次取值都让  $a[j]$  同  $a[k]$  比较,若  $a[j] < a[k]$  成立则把  $j$  的值赋给  $k$ ,使得  $a[k]$  始终为当前区间中已比较过的所有元素中的最小值,其中  $k$  的初值为  $i-1$ ,每次从当前排序区间选择出最小值  $a[k]$  后,都要把它与  $a[i-1]$  的值相交换,使得  $a[i-1]$  成为当前区间中的最小值。

根据以上分析,编写出此题的完整程序如下:

```
#include <iostream.h>
int a[10] = {42, 65, 80, 74, 36, 44, 28, 65, 94, 72}; // 在所有函数定义之外
// 定义数组或变量可以为在它们之后定义每个函数所使用
int n = 10; // 这里定义的数组 a 和 n 可为其后的两个函数所使用
void SelectSort() // 选择排序算法
{
    int i, j, k;
    for(i = 1; i < n; i++) { // 进行 n-1 次选择和交换
        k = i - 1; // 给 k 赋初值
        for(j = i; j < n; j++) // 选择出当前区间内的最小值 a[k]
            if(a[j] < a[k]) k = j;
        int x = a[i - 1]; a[i - 1] = a[k]; a[k] = x; // 交换 a[i - 1] 与 a[k] 的值
    }
}
void main()
{
    SelectSort(); // 调用函数对数组 a[n] 进行选择排序
    for(int i = 0; i < n; i++) cout << a[i] << ' ';
    // 依次输出数组 a[n] 中的每个元素值
    cout << endl;
}
```

该程序的运行结果为:

28 36 42 44 65 65 72 74 80 94

对数组  $a[10]$  中的元素进行选择排序的过程中,每一次选择和交换后各元素值的排列情况如图 4-1 所示,其中方括号内为下一次待排序区间,它是一个无序表,方括号前为已排好序的元素,它是一个有序表。

例 2. 已知 10 个常数如上例,请采用插入排序的方法对其进行排序并输出。

分析:插入排序方法的过程是,把数组  $a[n]$  中的  $n$  个元素看做为一个有序表和一个无序表,开始时有序表中只有一个元素  $a[0]$ ,无序表中包含有  $n-1$  个元素  $a[1] \sim a[n-1]$ ,以后每次从无序表中取出第一个元素  $a[i]$  ( $i=1, 2, \dots, n-1$ ),把它插入到前面有序表中的合适位置,使之仍为一个有序表,这样有序表就增加了一个元素,由上一次的  $a[0] \sim a[i-1]$  变为当前的  $a[0] \sim a[i]$ ,无序表中就减少了一个元素,由上一次的  $a[i] \sim a[n-1]$  变为当前的  $a[i+1] \sim a[n-1]$ ,经过  $n-1$  次插入过程后整个数组  $a$  中的  $n$  个元素就成为了一个有序表。

那么,如何在第  $i$  次把无序表中的第一个元素  $a[i]$  插入到前面有序表  $a[0] \sim a[i-1]$  中,

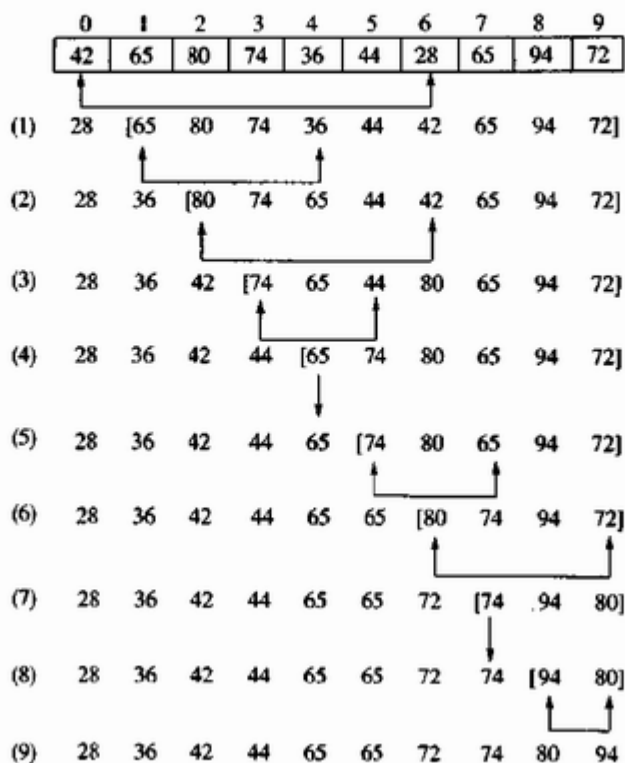


图 4-1 选择排序过程示例

使之成为一个新的有序表  $a[0] \sim a[i]$ 。具体做法是:从有序表的表尾元素  $a[i-1]$  开始,依次向前使每一个元素  $a[j]$  ( $j=i-1, i-2, \dots, 0$ ) 同  $x$  (用  $x$  暂存待插入元素  $a[i]$  的值) 进行比较,若  $x < a[j]$  则把  $a[j]$  后移一个位置,直到此条件不成立或  $j < 0$  为止,此时已空出的下标为  $j+1$  的位置就是  $x$  的插入位置,接着把  $x$  的值存入  $a[j+1]$  即可。

根据分析编写出程序如下:

```
#include <iostream.h>
int a[10] = {42, 65, 80, 74, 36, 44, 28, 65, 94, 72};
int n = 10;
void InsertSort() // 插入排序算法
{
    int i, j, x;
    for(i = 1; i < n; i++) { // 进行 n-1 次循环, 每次插入一个元素到有序表
        x = a[i]; // 将此次待插入元素存入 x
        for(j = i - 1; j >= 0; j--) // 为 x 顺序向前寻找插入位置
            if(x < a[j]) a[j+1] = a[j]; else break;
        a[j+1] = x; // 将 x 插入到已找到的插入位置
    }
}
void main()
{
    InsertSort(); // 调用插入排序算法对数组 a[n] 进行排序
}
```



```

    for(int i=0;i<n;i++) cout << a[i] << ' '; // 输出数组 a[n]
    cout << endl;
}

```

对数组  $a[10]$  中的元素进行插入排序的过程中,每次从无序表中取出第一个元素插入前面的有序表后各元素值的排列情况如图 4-2 所示,其中方括号内为本次得到的有序表,其后为无序表。

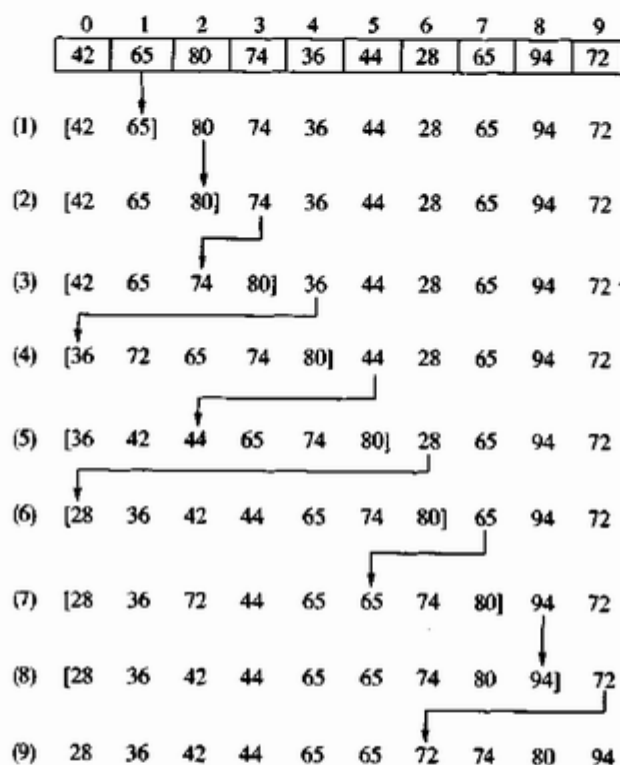


图 4-2 插入排序过程示例

#### 4.3.4 查找

例 1. 假定在一维数组  $a[10]$  中保存着 10 个整数 42, 55, 73, 28, 48, 66, 30, 65, 94, 72, 编一程序从中顺序查找出具有给定值  $x$  的元素, 若查找成功则返回该元素的下标位置, 否则表明查找失败返回 -1。

此程序比较简单, 假定把从一维数组中顺序查找的过程单独用一个函数模块来实现, 把调用该函数进行顺序查找通过主函数来实现, 则整个程序如下:

```

#include <iostream.h>
const int N=10; // 假定把数组中保存的整数个数用常量 N 表示
int a[N] = {42, 55, 73, 28, 48, 66, 30, 65, 94, 72};
int SequentialSearch(int x) // 顺序查找算法

```

```

{
    for(int i=0;i<N;i++)
        if(x==a[i]) return i; // 查找成功返回元素 a[i] 的下标值
    return -1; // 查找失败返回 -1
}

void main()
{
    int x1=48,x2=60,f;
    f=SequentialSearch(x1); // 从数组 a[N] 中查找值为 x1 的元素,
    // 将返回值赋给 f
    if(f== -1) cout << "查找" << x1 << "失败!" << endl;
    else cout << "查找" << x1 << "成功!" << "下标为" << f << endl;
    // 查找成功或失败分别显示出相应的信息
    f=SequentialSearch(x2); // 查找值为 x2 的元素, 返回值赋给 f
    if(f== -1) cout << "查找" << x2 << "失败!" << endl;
    else cout << "查找" << x2 << "成功!" << "下标为" << f << endl;
}

```

上机输入和运行该程序,得到的输出结果为:

```

查找 48 成功!下标为 4
查找 60 失败!

```

例 2. 假定一维数组  $a[N]$  中的  $N$  个元素是一个从小到大顺序排列的有序表,编一程序从  $a$  中二分查找出其值等于给定值  $x$  的元素。

分析:二分查找又称折半查找或对分查找。它比顺序查找要快得多,特别是当数据量很大时效果更显著。二分查找只能在有序表上进行,对于一个无序表则只能采用顺序查找。在有序表  $a[N]$  上进行二分查找的过程为:首先待查找区间为所有  $N$  个元素  $a[0] \sim a[N-1]$ ,将其中点元素  $a[mid]$  ( $mid = (N-1)/2$ ) 的值同给定值  $x$  进行比较,若  $x == a[mid]$  则表明查找成功,返回该元素的下标  $mid$  的值,若  $x < a[mid]$ ,则表明待查元素只可能落在该中点元素的左边区间  $a[0] \sim a[mid-1]$  中,接着只要在这个左边区间内继续进行二分查找即可,若  $x > a[mid]$ ,则表明待查元素只可能落在该中点元素的右边区间  $a[mid+1] \sim a[N-1]$  中,接着只要在这个右边区间内继续进行二分查找即可。这样经过一次比较后就使得查找区间缩小一半,如此进行下去,直到查找到对应的元素,返回下标值,或者查找区间变为空(即区间下界  $low$  大于区间上界  $high$ ),表明查找失败返回  $-1$  为止。

假定数组  $a[10]$  中的 10 个整型元素如下所示:

0	1	2	3	4	5	6	7	8	9
15	26	37	45	48	52	60	66	73	90

若要从中二分查找出值为 37 的元素,则具体过程为:开始时查找区间为  $a[0] \sim a[9]$ ,其中点元素的下标  $mid$  为 4,因  $a[4]$  的值为 48,其给定值 37 小于它,所以应接着在左区间  $a[0] \sim a[3]$  中继续二分查找,此时中点元素的下标  $mid$  为 1,因  $a[1]$  的值为 26,其给定值 37 大于它,所以应接着在右区间  $a[2] \sim a[3]$  中继续二分查找,此时中点元素的下标  $mid$  为  $(2+3)/2$  的值 2,因  $a[2]$  的值为 37,给定值与它相等,至此查找结束返回该元素的下标值 2。此查找过程可用图 4-3 表示出来,其中每次二分查找区间用方括号括起来,该区间的下界和上界分

别用 low 和 high 表示。

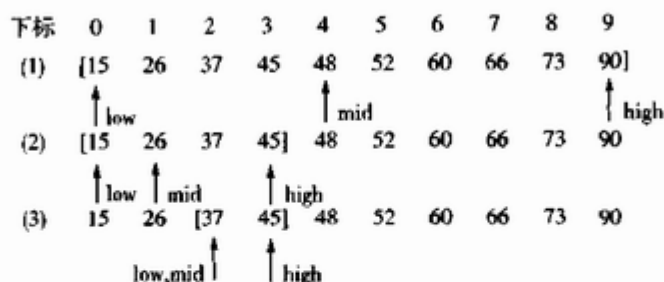


图 4-3 二分查找 37 的过程示意图

若要从数组 a[10]中二分查找其值为 70 的元素,则经过 3 次比较后因查找区间变为空,即区间下界 low 大于区间上界 high,所以查找失败,其查找过程如图 4-4 所示。

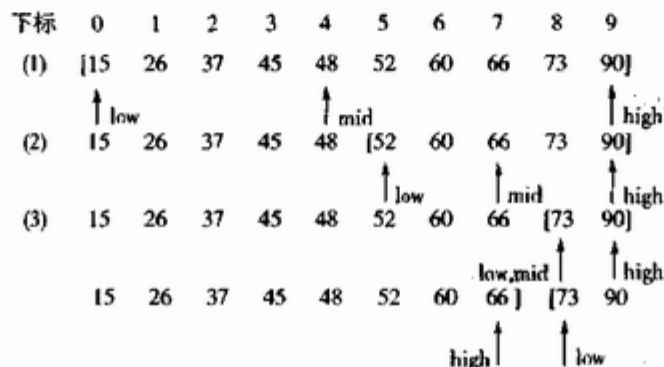


图 4-4 二分查找 70 的过程示意图

根据以上的分析和举例说明,编写出此题完整程序如下:

```
#include <iostream.h>
const int N=10; // 假定 N 等于 10
int a[N] = {15,26,37,45,48,52,60,66,73,90}; // 定义数组 a[N]并初始化
int BinarySearch(int x) // 二分查找算法
{
    int low=0, high=N-1; // 定义并初始化区间下界和上界变量
    int mid; // 定义保存中点元素下标的变量
    while(low <= high) { // 当当前查找区间非空时进行一次二分查找过程
        mid = (low+high)/2; // 计算出中点元素的下标
        if(x == a[mid]) return mid; // 查找成功返回
        else if(x < a[mid]) high = mid-1; // 修改 high 得到左区间
        else low = mid+1; // 修改 low 得到右区间
    }
    return -1; // 查找失败返回 -1
}

void main()
{
    int b[3] = {37,48,70}; // 假定待查元素值用数组 b 表示
```

```

int f; // 用于保存调用二分查找函数的返回值
for(int i=0;i<3;i++){
    f = BinarySearch(b[i]);
    if(f != -1)
        cout << "二分查找 " << b[i] << "成功!" << "下标为 " << f << endl;
    else cout << "二分查找 " << b[i] << "失败!" << endl;
}
}

```

该程序运行结果如下:

```

二分查找 37 成功:下标为 2
二分查找 48 成功:下标为 4
二分查找 70 失败:

```

## 4.4 字符串

### 4.4.1 字符串概念

#### 1. 字符串的定义

在C++语言中,一个字符串就是用一对双引号括起来的一串字符,其双引号是该字符串的起、止标志符,它不属于字符串本身的字符。如:

- (1) "string "
- (2) "Visual C++ "
- (3) "a + b = "
- (4) "姓名,年龄 "
- (5) "Input a integer to x: "

都是C++字符串。

一个字符串的长度等于双引号内所有字符的长度之和,其中每个ASCII码字符的长度为1,每个区位码字符(如汉字)的长度为2。如上面每个字符串的长度依次为6,10,4,9和21。

特殊地,当一个字符串不含有任何字符时,则称为空串,其长度为0,当只含有一个字符时,其长度为1。如""是一个空串,"A"是一个长度为1的字符串。注意:'A'和"A"是不同的,前者表示一个字符A,后者表示一个字符串A,虽然它们的值都是A,但稍后便知它们具有不同的存储格式。

在一个字符串中不仅可以一般字符,而且可以使用转义字符。如字符串"\\"cout << ch\\"n"中包含有11个字符,其中第1个和第10个为表示双引号的转义字符,最后一个为表示换行的转义字符。

#### 2. 字符串的存储

在C++语言中,存储字符串是利用一维字符数组来实现的,该字符数组的长度要大于

等于待存字符串的长度加1。设一个字符串的长度为  $n$ , 则用于存储该字符串的数组的长度应至少为  $n+1$ 。

把一个字符串存入到数组时, 是把每个字符依次存入到数组中对应元素中, 即把第一个字符存入到下标为0的元素中, 第二个字符存入到下标为1的元素中, 依次类推, 最后把一个空字符 '\0' 存入到下标为  $n$  的元素中, 这里假定字符串的长度为  $n$ 。当然存储每个字符就是存储它的 ASCII 码或区位码。如利用一维字符数组  $a[12]$  来存储字符串 "Strings. \n" 时, 数组  $a$  中的内容为:

	0	1	2	3	4	5	6	7	8	9	10	11
字符表示:	S	t	r	i	n	g	s	.	\n	\0		
ASCII 码表示:	83	116	114	105	110	103	115	46	10	0		

若一个数组被存储了一个字符串后, 其尾部还有剩余的元素, 实际上也被自动存储上空字符 '\0'。在上述例子中,  $a[10]$  和  $a[11]$  元素的值也被自动置为 '\0'。

### 3. 利用字符串初始化字符数组

一个字符串能够在定义字符数组时作为初始化数据被存入到数组中, 不能通过赋值表达式直接赋值。如:

- (1) `char a[10] = "array";`
- (2) `char b[20] = "This is a pen. ";`
- (3) `char c[8] = " ";`
- (4) `a = "struct";`
- (5) `a[0] = 'A';`

第一条语句定义了字符数组  $a[10]$  并被初始化为 "array", 其中  $a[0] \sim a[5]$  元素的值依次为字符 'a', 'r', 'r', 'a', 'y' 和 '\0'; 第二条语句定义了字符数组  $b[20]$ , 其中  $b[i]$  元素 ( $0 \leq i \leq 13$ ) 被初始化为所给字符串中的第  $i+1$  个字符,  $b[14]$  被初始化为字符串结束标志符 '\0'; 第三条语句定义了一个字符数组  $c[8]$  并初始化为一个空串, 此时它的每个元素的值均为 '\0'; 第四条语句是非法的, 因为它试图使用赋值号把一个字符串直接赋值给一个数组, 这在 C++ 中是不允许的; 第五条是合法的, 它把字符 'A' 赋给了  $a[0]$  元素, 使得数组  $a$  中保存的字符串变为 "Array"。

利用字符串初始化字符数组也可以写成初值表的方式。如上述第一条语句与下面语句完全等效。

`char a[10] = {'a', 'r', 'r', 'a', 'y', '\0'}; // '\0' 也可直接写为 0`

注意: 最后一个字符 '\0' 是必不可少的, 它是一个字符串在数组中结束的标志。

### 4. 字符串的输入和输出

用于存储字符串的字符数组, 其元素可以通过下标运算符访问, 这与一般字符数组和其他任何类型的数组是相同的。除此之外, 还可以对它进行整体输入输出操作和有关的函数

操作。如假定 `a[11]` 为一个字符数组,则:

(1) `cin >> a;`

(2) `cout << a;`

是允许的,即允许在提取或插入操作符后面使用一个字符数组名实现向数组输入字符串或输出数组中保存的字符串的目的。

计算机执行上述第一条语句时,要求用户从键盘上输入一个不含空格的字符串,用空格或回车键作为字符串输入的结束符,系统就把该字符串存入到字符数组 `a` 中,当然在存入的整个字符串的后面将自动存入一个结束符 `'\0'`。

注意:输入的字符串的长度要小于数组 `a` 的长度,这样才能够把输入的字符串有效地存储起来,否则是程序设计的一个逻辑错误,可能导致程序运行出错。另外,输入的字符串不需要另加双引号定界符,只要输入字符串本身即可,假如输入了双引号则被视为一般字符。

执行上述第二条语句时向屏幕输出在数组 `a` 中保存的字符串,它将从数组 `a` 中下标为 0 的元素开始,依次输出每个元素的值,直到碰到字符串结束符 `'\0'` 为止。若数组 `a` 中的内容为:

0	1	2	3	4	5	6	7	8	9	10
w	r	i	t	e	\0	r	e	a	d	\0

则输出 `a` 时只会输出第一个空字符前面的字符串 `"write"`,而它后面的任何内容都不会被输出。

利用插入操作符 `<<` 不仅能够输出字符数组中保存的字符串,而且能够直接输出一个字符串常量,即用双引号括起来的字符串。如:

```
cout << "x+y = " << x+y << endl;
```

此语句输出字符串 `"x+y = "` 后接着输出 `x+y` 的值和一个换行符。若 `x` 和 `y` 的值分别为 15 和 24,则得到的输出结果为:

```
x+y = 39
```

## 5. 利用二维数组存储字符串

利用一维字符数组能够保存一个字符串,而利用二维字符数组能够同时保存若干个字符串,最多能保存的字符串个数等于该数组的行数。如:

(1) `char a[7][4] = { "SUN ", "MON ", "TUE ", "WED ", "THU ", "FRI ", "SAT "};`

(2) `char b[][8] = { "well ", "good ", "middle ", "pass ", "bad "};`

(3) `char c[6][10] = { "int ", "double ", "char "};`

(4) `char d[10][20] = { " "};`

在第一条语句中定义了一个二维字符数组 `a`,它包含 7 行,每行具有 4 个字符空间,每行用来保存长度小于等于 3 的一个字符串。该语句同时对 `a` 进行了初始化,使得 `"SUN"` 被保存到行下标为 0 的行里,该行包含 `a[0][0]`, `a[0][1]`, `a[0][2]` 和 `a[0][3]` 这四个二维元素,每

个元素的值依次为'S','U','N'和'\0',同样"MON"被保存到行下标为1的行里,……,"SAT"被保存到行下标为6的行里。以后既可以利用双下标变量  $a[i][j]$  ( $0 \leq i \leq 6, 0 \leq j \leq 2$ ) 访问每个字符元素,也可以利用只带行下标的单下标变量  $a[i]$  ( $0 \leq i \leq 6$ ) 访问每个字符串。如  $a[2]$  则表示字符串"TUE",  $a[5]$  则表示字符串"FRI",  $\text{cin} \gg a[4]$  则表示从键盘上向  $a[4]$  输入一个字符串,  $\text{cout} \ll a[i]$  则表示向屏幕输出  $a[i]$  中保存的字符串。

上述第二条语句定义了一个二维字符数组  $b$ , 它的行数没有显式地给出, 隐含为初值表中所列字符串的个数, 因所列字符串为5个, 所以该数组  $b$  的行数为5, 又因列数被定义为8, 所以每一行所存字符串的长度要小于等于7。该语句被执行后,  $b[0]$  表示字符串"well",  $b[1]$  表示字符串"good"……。

第三条语句定义了一个二维字符数组  $c$ , 它最多能够存储6个字符串, 每个字符串的长度要不超过9, 该数组前三个字符串元素  $c[0]$ ,  $c[1]$  和  $c[2]$  分别被初始化为"int", "double" 和 "char", 后三个字符串元素均被初始化为空串。

第四条语句定义了一个能够存储10个字符串的二维字符数组  $d$ , 每个字符串的长度不得超过19。该语句对所有字符串元素初始化为一个空串。

下面的程序段能够从键盘上依次输入10个字符串到二维字符数组  $w$  中保存起来, 输入的每个字符串的长度不得超过29。

```
const int N=10;
char w[N][30];
for(int i=0;i<N;i++) cin >> w[i];
```

下面的一条 for 语句将按相反的次序依次输出在数组  $w$  中保存的所有字符串, 在输出每个字符串之后都输出一个换行符。

```
for(i=N-1;i>=0;i--) cout << w[i] << endl;
```

#### 4.4.2 字符串函数

C++ 系统专门为处理字符串提供了一些预定义函数供编程者使用, 这些函数的原型被保存在 `string.h` 头文件中, 当用户在程序文件开始使用 `#include` 命令把该头文件引入之后, 就可以在后面定义每个函数中调用这些预定义的字符串函数, 对字符串作相应的处理。

C++ 系统提供的处理字符串的预定义函数有许多, 从 C++ 库函数资料中可以得到全部说明, 下面简要介绍其中几个主要的字符串函数。

##### 1. 求字符串长度

函数原型: `int strlen(const char s[]);`

此函数只有一个参数, 它是一个元素类型为字符的数组参数, 它前面使用的保留字 `const` 表示该参数的内容在函数体中是不允许改变的, 当然使用它不影响读取参数的值。该函数对应的实参可以为任何形式的字符串, 如可以是一个字符串常量, 可以是一个一维字符数组名, 也可以是二维字符数组中只带行下标的单下标变量。待学习完下一章指针之后, 读者将会对数组参数有更深刻的理解。

调用该函数时,将返回实参字符串的长度。

例如,假定一个字符数组 `a[10]` 的内容为 "", `b[10]` 的内容为 "a", `c[20]` 的内容为 "StringLength", 则 `strlen(a)`, `strlen(b)` 和 `strlen(c)` 的值分别为 0, 1 和 12。

若要计算字符串常量 "constant" 的长度,则使用 `strlen("constant")` 即可得到,返回值为 8。

## 2. 字符串拷贝

函数原型: `char * strcpy(char * dest, const char * src);`

此函数有两个参数,它们都是字符指针参数。因为每个字符指针是指向相应字符串的首地址,而字符数组名就是所存字符串的首地址,所以字符数组名也就是一个字符指针。字符指针参数说明同字符数组参数说明是等价的,也就是说,该函数中的两个参数说明分别同 `char dest[]` 和 `const char src[]` 是等价的。无论采用哪一种说明, `dest` 或 `src` 都能够接受调用时由实参传送来的一个字符指针,即一个字符串存储空间的首地址。

该函数的功能是把第二个参数 `src` 所指字符串拷贝(即赋值)到第一个参数 `dest` 所指的存储空间(即 `dest` 字符数组)中,然后返回 `dest` 的值,它是一个字符指针。

因为该函数只需要从 `src` 字符串中读取内容,不需要修改它,所以用 `const` 修饰,而对于第一个参数 `dest`,需要修改它的内容,所以就不能用 `const` 修饰。

关于指针的更详细的内容将在下一章讨论。

请看下面的程序段:

```
char a[10], b[10] = "copy";
strcpy(a, b);
cout << a << ' ' << b << ' ';
cout << strlen(a) << ' ' << strlen(b) << endl;
```

该程序段首先定义了两个字符数组 `a` 和 `b`, 并对 `b` 初始化为 "copy"; 接着调用 `strcpy` 函数, 把 `b` 所指向(即数组 `b` 保存)的字符串 "copy" 拷贝到 `a` 所指向(即数组 `a` 占用)的存储空间中, 使得数组 `a` 保存的字符串同样为 "copy", 该函数返回 `a` 的值被自动丢失; 该程序段中的第三条语句输出 `a` 和 `b` 所指向的字符串, 或者说输出数组 `a` 和 `b` 中所保存的字符串; 第四条语句输出 `a` 和 `b` 所指向的字符串的长度。该程序段的运行结果为:

```
copy copy 4 4
```

## 3. 字符串连接

函数原型: `char * strcat(char * dest, const char * src);`

此函数同上述 `strcpy` 函数具有完全相同的参数说明和返回值类型。函数功能是把第二个参数 `src` 所指字符串拷贝到第一个参数 `dest` 所指字符串之后的存储空间中, 或者说, 把 `src` 所指字符串连接到 `dest` 所指的字符串之后。该函数返回 `dest` 的值。

使用该函数时要确保 `dest` 所指字符串之后有足够的存储空间用于存储 `src` 串。

调用此函数之后, 第一个实参所指字符串的长度将等于两个实参所指字符串的长度之和。

例如:



```

char a[20] = "string";    // 字符串长度为 6
char b[] = "catenation"; // 字符串长度为 10
strcat(a, " ");          // 连接一个空格到 a 串之后
strcat(a, b);            // 把 b 串连接到 a 串之后
cout << a << ' ' << strlen(a) << endl;

```

执行该程序段得到的输出结果为:

```
string catenation 17
```

#### 4. 字符串比较

函数原型: `int strcmp(const char* s1, const char* s2);`

此函数带有两个字符指针参数,各自指向相应的字符串,函数的返回值为整型。

该函数的功能为:比较 `s1` 所指字符串与 `s2` 所指字符串的大小,若 `s1` 串大于 `s2` 串则返回一个大于 0 的值,在 C++ 6.0 中返回 1;若 `s1` 串等于 `s2` 串则返回值为 0;若 `s1` 串小于 `s2` 串则返回一个小于 0 的值,在 C++ 6.0 中返回 -1。

比较 `s1` 串和 `s2` 串的大小是一个循环过程,需要从两个串的第一个字符起依次向后比较,整个比较过程可用下面的程序段描述出来。

```

int i;
for(i = 0; s1[i] && s2[i]; i++)
    // 循环的正常结束要等到任一个字符串中的字符比较完
    if(s1[i] > s2[i]) return 1;
    else if(s1[i] < s2[i]) return -1;
if(s1[i] == 0 && s2[i] == 0) return 0;
    // 等于号右边的数值 0 可改为 '\0'
else if(s1[i] != 0) return 1;
else return -1;

```

在这个程序段中使用的 `s1[i]` 和 `s2[i]` 分别为 `s1` 数组和 `s2` 数组中下标为 `i` 的元素,分别表示 `s1` 和 `s2` 所指字符串中的第 `i+1` 个字符。

假定字符数组 `a`, `b` 和 `c` 的值分别为字符串 "1234", "4321", "1304", 则:

```

strcmp(a, "1234") = 0      strcmp(a, b) = -1
strcmp(a, c) = -1         strcmp(a, "123") = 1
strcmp("A", "a") = -1     strcmp("英文", "汉字") = 1

```

#### 5. 从字符串中查找字符

函数原型: `char* strchr(const char* s, int c);`

该函数从 `s` 所指字符串中的第一个字符起顺序查找 ASCII 码为 `c` 值的字符,若查找成功则返回该字符的存储地址,否则返回 NULL(即数值 0)。一个字符数组 `a` 中下标为 `i` 的元素 `a[i]` 的存储地址就是所存字符的存储地址,此地址可根据 `a + sizeof(char) * i` 计算出来,因为 `sizeof(char) = 1`,所以 `a[i]` 的地址为 `a + i` 的值。

当调用该函数时传送给第二个形参 `c` 的实参,可以为整数,但通常是一个待查找的字符。

例如,进行 `strchr("abcd", 'c')` 函数调用将返回字符串 "abcd" 的首地址加 2 的值,进行 `strchr("abcd", 'e')` 函数调用将返回地址值 `NULL`。

#### 6. 从字符串中逆序查找字符

函数原型: `char * strrchr(const char * s, int c);`

它与上面介绍的 `strchr` 函数功能相同,都是从字符串中查找字符,但查找次序不同,该函数是从 `s` 所指字符串的最后一个字符起顺序向前查找,同样若查找成功返回字符的存储地址,否则返回 `NULL`。

例如,进行 `strrchr("abcab", 'a')` 函数调用将返回字符串 "abcab" 的首地址加 3 的值,若把函数名改为 `strchr`,则结果为 "abcab" 的首地址值。

#### 7. 从字符串中查找子串

函数原型: `char * strstr(const char * s1, const char * s2);`

该函数从第一个参数 `s1` 所指字符串中第一个字符起,顺序向后查找出与第二个参数 `s2` 所指字符串相同的子串,若查找成功则返回该子串的首地址,否则返回 `NULL`。

例如:

```
char a[20] = "abcdabxcdabxy";
char b[4] = "abx";
char c[4] = "axy";
cout << strstr(a,b) << endl;
if(strstr(a,c) == NULL)
    cout << "Not found!" << endl;
cout << strstr("学习文化知识", "文化") << endl;
```

该程序段首先定义了三个字符数组并分别进行了初始化,接着输出以 `strstr(a,b)` 的返回地址为首地址的字符串,该字符串为 "abxcdabxy",执行第四条 `if` 语句时,因从 `a` 串中查找不到 `c` 串,所以判断表达式成立,将向屏幕输出字符串 "Not found!" 和一个换行符,执行最后一条语句时,输出的字符串为 "文化知识"。

### 4.4.3 字符串应用举例

1. 编写一个程序,首先从键盘上输入一个字符串,接着输入一个字符,然后分别统计出字符串中大于、等于、小于该字符的个数。

分析:设用于保存输入字符串的字符数组用 `a[N]` 表示,用于保存一个输入字符的变量用 `ch` 表示,用于分三种情况进行统计的计数变量分别用 `c1`, `c2` 和 `c3` 表示。定义字符数组所使用的 `N` 为一个需事先定义的整型常量,它要大于输入的字符串的长度。

下面是此题的一个完整程序。

```
#include <iostream.h>
const N=30; // 假定输入的字符串的长度小于 30
void main()
{
```

```

char a[N],ch;
int c1,c2,c3;
c1=c2=c3=0;
cout << "输入一个字符串: ";
cin >> a;
cout << "输入一个字符: ";
cin >> ch;
int i=0;
while(a[i]) { // 统计
    if(a[i]>ch) c1++;
    else if(a[i]==ch) c2++;
    else c3++;
    i++;
}
cout << "c1 = " << c1 << endl;
cout << "c2 = " << c2 << endl;
cout << "c3 = " << c3 << endl;
}

```

2. 编一程序,首先输入 10 个字符串到一个二维字符数组中,接着输入一个待查的字符串,然后从二维字符数组中查找统计出含有待查字符串的个数。

此程序比较简单,编写如下:

```

#include <iostream.h>
#include <string.h>
void main()
{
    char a[10][30] = {" "};
    // 用于存储 10 个字符串,假定每个串的长度小于 30
    char s[30]; // 存储待查的字符串
    int i,k=0;
    cout << "输入 10 个字符串: ";
    for(i=0;i<10;i++) cin >> a[i];
    cout << "输入一个待查字符串: ";
    cin >> s;
    for(i=0;i<10;i++)
        if(strcmp(a[i],s)==0) k++;
    cout << "字符串个数: " << k << endl;
}

```

3. 编一程序,首先输入 M 个字符串到一个二维字符数组中,并假定每个字符串的长度均小于 N,M 和 N 为事先定义的整型常量,接着对这 M 个字符串进行选择排序,最后输出排序结果。

分析:我们已经在前面学习了对简单类型的数据进行选择排序的方法和算法描述,在这里只要把它移植过来用于字符串排序即可,不过对字符串的比较和赋值必须使用字符串比较和拷贝函数来实现。

此题的完整程序如下:

```

#include <iostream.h>

```

```

#include<string.h>
const M=10, N=30; // 定义常量 M 和 N
void SelectSort(char a[M][N]) // 对字符串进行选择排序算法
{
    int i,j,k;
    for(i=1;i<M;i++){ // 进行 M-1 次选择和交换
        // 给 k 赋初值
        k=i-1;
        // 选择出当前区间内的最小值 a[k]
        for(j=i;j<M;j++){
            if(strcmp(a[j],a[k])<0) k=j; // 进行字符串比较
        }
        // 定义字符数组 x 用于交换 a[i-1] 和 a[k] 的值
        char x[N];
        // 利用字符串拷贝函数交换 a[i-1] 与 a[k] 的值
        strcpy(x,a[i-1]);
        strcpy(a[i-1],a[k]);
        strcpy(a[k],x);
    }
}

void main()
{
    // 定义二维字符数组 b 并初始化每个字符串为空串
    char b[M][N]={" "};
    // 从键盘输入 M 个字符串到字符串数组 b 中
    int i;
    cout << "输入 " << M << " 个字符串: ";
    for(i=0;i<M;i++) cin >> b[i];
    // 调用字符串选择排序算法对字符串数组 b 进行选择排序
    SelectSort(b);
    // 依次输出字符串数组 b 中的每个字符串
    for(i=0;i<M;i++) cout << b[i] << endl;
}

```

4. 编一程序,首先定义二维字符数组  $ax[M][N]$  和一维整型数组  $bx[M]$ ,接着从键盘上依次输入 M 个人的姓名和成绩,每次输入的姓名和成绩分别存入到  $ax[i][N]$  和  $bx[i]$  中,其中  $0 \leq i \leq M-1$ ,然后调用选择排序算法按照成绩从高到低的次序排列  $ax$  和  $bx$  数组中的元素,最后按照成绩从高到低的次序输出每个人的姓名和成绩。

根据以往所学的知识,可以编写出完整程序如下:

```

#include<iomanip.h>
#include<string.h>
const M=10, N=30; // 定义常量 M 和 N
void SelectSort(char a[M][N], int b[M])
// 算法中对数组参数的操作就是对相应实参数组的操作,
// 待学习了后面的函数一章后才会有深刻的理解。
{
    int i,j,k;
    for(i=1;i<M;i++){ // 进行 M-1 次选择和交换
        // 给 k 赋初值
        k=i-1;

```

```

        // 选择出当前区间内的最大值 b[k]
        for(j = i; j < M; j++)
            if(b[j] > b[k]) k = j;
        // 交换 a[i-1] 和 a[k], b[i-1] 和 b[k] 的值, 使成绩和姓名同步被交换
        char x[N]; int y;
        strcpy(x, a[i-1]); strcpy(a[i-1], a[k]); strcpy(a[k], x);
        y = b[i-1]; b[i-1] = b[k]; b[k] = y;
    }
}

void main()
{
    // 定义 ax 和 bx 数组
    char ax[M][N];
    int bx[M];
    // 从键盘输入 M 个人的姓名和成绩到数组 ax 和 bx 中
    int i;
    cout << "输入 " << M << " 个人的姓名和成绩: ";
    for(i = 0; i < M; i++) cin >> ax[i] >> bx[i];
    // 调用选择排序算法对 ax 和 bx 数组按成绩进行选择排序
    SelectSort(ax, bx);
    // 此语句使以后的输出结果按在给定宽度内左对齐显示,
    // 默认情况是按右对齐显示
    cout.setf(ios::left);
    // 按排序结果依次输出每个人的姓名和成绩
    for(i = 0; i < M; i++)
        cout << setw(30) << ax[i] << setw(4) << bx[i] << endl;
}

```

下面是程序的一次运行结果:

```

输入 10 个人的姓名和成绩:
wer 76 erty 93 asdf 54 wqert 80 dwerty 65
zxc 88 vcfdsjkh 58 gfhj 42 sfr 74 jkzh 86
erty          93
zxc           88
jkzh          86
wqert         80
wer           76
sfr           74
dwerty        65
vcfdsjkh      58
asdf          54
gfhj          42

```

## 习题四

### (一) 填空题

1. 数组元素  $a[i]$  是该数组中的第\_\_\_\_\_个元素。

2. 元素类型为 int 的数组 a[10] 共占用\_\_\_\_\_字节的存储空间, 其中 a[5] 元素的字节地址为\_\_\_\_\_。
3. 元素类型为 double 的二维数组 a[4][6] 共占用\_\_\_\_\_字节的存储空间, 其中 a[2][5] 元素的字节地址为\_\_\_\_\_。
4. 元素类型为 char 的二维数组 a[10][30] 共占用\_\_\_\_\_字节的存储空间, 其中 a[3][4] 元素的字节地址为\_\_\_\_\_。
5. 假定对数组 a[] 进行初始化的数据为 {2, 7, 9, 6, 5, 7, 10}, 则 a[2] 和 a[5] 分别被初始化为\_\_\_\_\_和\_\_\_\_\_。
6. 假定对二维数组 a[3][4] 进行初始化的数据为 { {3, 5, 6}, {2, 8}, {7} }, 则 a[0][0], a[1][1] 和 a[2][3] 分别被初始化为\_\_\_\_\_, \_\_\_\_\_和\_\_\_\_\_。
7. 存储字符 'a' 和字符串 "a" 分别需要占用\_\_\_\_\_和\_\_\_\_\_个字节。
8. 空串的长度为\_\_\_\_\_, 存储它需要占用\_\_\_\_\_个字节。
9. 字符串 "\'a\'xy=4\n" 的长度为\_\_\_\_\_。
10. 字符串 "a: \\ xk \\ 数据" 的长度为\_\_\_\_\_。
11. 对于一个长度为 n 的字符串, 需要占用\_\_\_\_\_个字节, 用于存储该字符串的字符数组的长度至少为\_\_\_\_\_。
12. 若 a 是一个字符数组, 则 cin >> a 表示从键盘上读入一个\_\_\_\_\_到数组 a 中, cout << a 表示向屏幕输出在 a 中所存的\_\_\_\_\_。
13. 一个二维字符数组 a[10][20] 能够存储\_\_\_\_\_个字符串, 每个字符串的长度至多为\_\_\_\_\_。
14. 对一个一维字符数组 a 进行初始化的数据为 "12345", 则 a[0] 和 a[3] 元素对应的初值分别为\_\_\_\_\_和\_\_\_\_\_。
15. 对一个二维字符数组 a 进行初始化的数据为 { "123", "456", "789" }, 则 a[1] 和 a[4] 元素对应的初始字符串分别为\_\_\_\_\_和\_\_\_\_\_。
16. strlen("apple") 的值为\_\_\_\_\_, strcmp("a", "A") 的值为\_\_\_\_\_。
17. 假定字符数组 a 中保存的字符串为 "abc", 则调用 strcat(a, "def") 后 a 中保存的字符串变为\_\_\_\_\_。
18. 若需要把一个字符串 "aaa" 赋值到字符数组 a 中, 则需要执行\_\_\_\_\_的函数调用实现。
19. 假定字符数组 a 中的所存字符串为 "abedbcdf", 则 strchr(a, 'c') 的值比 strchr(a, 'c') 的值小\_\_\_\_\_。
20. 假定字符数组 a 中的所存字符串为 "abedbcdf", 则 strstr(a, 'cd') 的返回值比 a 的值大\_\_\_\_\_。
21. 使用 "type int Integer;" 语句把\_\_\_\_\_定义为\_\_\_\_\_的别名, 以后使用 Integer 定义的对象具有\_\_\_\_\_类型。
22. 使用 "type int AA[10];" 语句定义\_\_\_\_\_为含有\_\_\_\_\_个\_\_\_\_\_型元素的数组类型。
23. 使用 "type char BB[10][50];" 语句定义\_\_\_\_\_为含有\_\_\_\_\_行\_\_\_\_\_列共\_\_\_\_\_个\_\_\_\_\_型元素的数组类型。

24. 使用“type int CC[4][6];”语句之后,再使用“CC a;”语句定义了 a 为一个含有\_\_\_\_\_行\_\_\_\_\_列共\_\_\_\_\_个\_\_\_\_\_型元素的二维数组。

(二) 写出下列程序运行后的输出结果并上机验证

```
1. #include<iostream.h>
void main() {
    int a[10] = {12,39,26,41,55,63,72,40,83,95};
    int i1 = 0, i2 = 0;
    for(int i = 0; i < 10; i++)
        if(a[i] % 2 == 1) i1++; else i2++;
    cout << i1 << ' ' << i2 << endl;
}
```

```
2. #include<iostream.h>
void main() {
    int a[8] = {36,25,48,14,55,40,32,66};
    int b1, b2;
    b1 = b2 = a[0];
    for(int i = 1; i < 8; i++)
        if(a[i] > b1) {
            if(b1 > b2) b2 = b1;
            b1 = a[i];
        }
    cout << b1 << ' ' << b2 << endl;
}
```

```
3. #include<iostream.h>
void main() {
    int a[8] = {36,25,48,14,55,40,32,66};
    int b1, b2;
    b1 = b2 = a[0];
    for(int i = 1; i < 8; i++)
        if(a[i] < b1) {
            if(b1 < b2) b2 = b1;
            b1 = a[i];
        }
    cout << b1 << ' ' << b2 << endl;
}
```

```
4. #include<iostream.h>
void main() {
    char a[] = "abcdabcaabfgacd";
    int i1 = 0, i2 = 0, i = 0;
    while(a[i]) {
        if(a[i] == 'a') i1++;
        if(a[i] == 'b') i2++;
        i++;
    }
    cout << i1 << ' ' << i2 << ' ' << i << endl;
}
```

5. 

```
#include <iostream.h>
void main() {
    char a[] = "abcdabcbadbaeff ";
    int b[5] = {0}, i = 0;
    while(a[i]) {
        switch(a[i]) {
            case 'a': b[0] ++; break;
            case 'b': b[1] ++; break;
            case 'c': b[2] ++; break;
            case 'd': b[3] ++; break;
            default: b[4] ++;
        }
        i ++;
    }
    for(i = 0; i < 5; i ++ ) cout << b[i] << ' ';
    cout << endl;
}
```
6. 

```
#include <iostream.h>
void main() {
    int a[10] = {76,83,54,62,40,75,90,92,77,84};
    int b[4] = {60,70,90,101};
    int c[4] = {0};
    for(int i = 0; i < 10; i ++ ) {
        int j = 0;
        while(a[i] >= b[j]) j ++;
        c[j] ++;
    }
    for(i = 0; i < 4; i ++ ) cout << c[i] << ' ';
    cout << endl;
}
```
7. 

```
#include <iostream.h>
void main() {
    int a[3][4] = {{1,2,7,8},{5,6,11,12},{9,10,3,4}};
    int m = a[0][0];
    int ii = 0, jj = 0;
    for(int i = 0; i < 3; i ++ )
        for(int j = 0; j < 4; j ++ )
            if(a[i][j] > m) {m = a[i][j]; ii = i; jj = j;}
    cout << ii << ' ' << jj << ' ' << a[ii][jj] << endl;
}
```
8. 

```
#include <iostream.h>
#include <string.h>
void main() {
    char a[5][10] = {"student", "worker", "cadre", "soldier", "peasant"};
    char s1[10], s2[10];
    strcpy(s1, a[0]); strcpy(s2, a[0]);
    for(int i = 1; i < 5; i ++ ) {
        if(strcmp(a[i], s1) > 0) strcpy(s1, a[i]);
    }
}
```



```

        if(strcmp(a[i], s2) < 0) strcpy(s2, a[i]);
    }
    cout << s1 << ' ' << s2 << endl;
}

```

```

9. #include <iostream.h>
const int N=10;
int BinarySearch(int a[N],int x)
{
    int low=0, high=N-1;
    int mid;
    while(low <= high) {
        mid = (low+high)/2;
        if(x == a[mid]) return mid;
        else if(x < a[mid]) high = mid - 1;
        else low = mid + 1;
    }
    return -1;
}
void main()
{
    int b[N] = {15,26,37,45,48,52,60,66,73,90};
    int c[4] = {48,26,73,80};
    int i;
    for(i=0;i<4;i++)
        cout << BinarySearch(b,c[i]) << ' ';
}

```

### (三) 指出下列每个函数的功能

1. void f1(int a[], int n)
 

```

{
    for(int i=0;i<n/2;i++) {
        int x=a[i];
        a[i]=a[n-i-1];
        a[n-i-1]=x;
    }
}

```
2. void f2(int a[], int n)
 

```

{
    int i; double sum=0;
    for(i=0;i<n;i++) sum += a[i];
    sum/=n;
    for(i=0;i<n;i++)
        if(a[i] >= sum) cout << a[i] << ' ';
    cout << endl;
}

```
3. void f3(char a[])
 

```

{

```

```

int i,c[5] = {0};
for(i=0;a[i];i++)
    switch(a[i]) {
        case ',': c[0]++;break;
        case ';': c[1]++;break;
        case '(':
        case ')': c[2]++;break;
        case '{':
        case '}': c[3]++;break;
        case '[':
        case ']': c[4]++;break;
    }
for(i=0;i<5;i++) cout << c[i] << ' ';
cout << endl;
}

```

```

4. void f4(char a[M][N])
{
    int c1,c2,c3;
    c1=c2=c3=0;
    for(int i=0;i<M;i++)
        if(strlen(a[i])<5) c1++;
        else if(strlen(a[i])>=5 && strlen(a[i])<15) c2++;
        else c3++;
    cout << c1 << ' ' << c2 << ' ' << c3 << endl;
}

```

#### (四) 编写下列程序并上机运行

1. 有一个数列,它的第一项为 0,第二项为 1,以后每一项都是它的前两项之和,试产生出此数列的前 20 项,并按逆序显示出来。

2. 从键盘上输入一个字符串,假定该字符串的长度不超过 30,试统计出该串中所有十进制数字字符的个数。

3. 首先从键盘上输入一个 4 行 4 列的一个实数矩阵到一个二维数组中,然后求出主对角线上元素之乘积。

4. 已知一个数值矩阵为  $\begin{bmatrix} 3 & 8 & 2 & 9 \\ 4 & 7 & 3 & 6 \\ 5 & 2 & 8 & 4 \end{bmatrix}$ , 求出该矩阵的转置矩阵并输出出来,其中转置矩阵中的  $[i][j]$  位置上的元素等于原矩阵中的  $[j][i]$  位置上的元素。

5. 已知一个矩阵 A 为  $\begin{bmatrix} 3 & 0 & 4 & 5 \\ 6 & 2 & 1 & 7 \\ 4 & 1 & 5 & 8 \end{bmatrix}$ , 另一个矩阵 B 为  $\begin{bmatrix} 1 & 4 & 0 & 3 \\ 2 & 5 & 1 & 6 \\ 0 & 7 & 4 & 4 \\ 9 & 3 & 6 & 0 \end{bmatrix}$ , 求出 A 与 B 的乘

积矩阵 C[3][4]并输出出来,其中 C 中的每个元素  $C[i][j]$  等于  $\sum_{k=0}^3 A[i][k] * B[k][j]$ 。

6. 首先让计算机随机产生出 10 个两位正整数,然后按照从小到大的次序显示出来。

7. 从键盘上输入一个字符串,假定字符串的长度小于 80,试分别统计出每一种英文字

母(大、小写等同看待)的个数并输出出来。

8. 某学校有 12 名学生参加 100 米短跑比赛,每个运动员号和成绩如 4-4 所示,请按照比赛成绩排名并输出,要求每一行输出名次、运动员号和比赛成绩三项数据。

表 4-4 100 米短跑比赛成绩

运动员号	成绩(秒)	运动员号	成绩(秒)
001	13.6	031	14.9
002	14.8	036	12.6
010	12.0	037	13.4
011	12.7	102	12.5
023	15.6	325	15.3
025	13.4	438	12.7

## 第五章 指 针

### 5.1 指针的概念

指针就是内存单元的地址。每个内存单元为二进制的八位,即一个字节。每个内存单元对应着一个编号(即地址),计算机控制器就是通过这个地址访问(即存取)对应单元中的内容。

一种类型的数据占用内存中固定个数的存储单元,如 char 型数据(即字符)占用一个存储单元, int 型整数占用 4 个存储单元,即 4 个字节, double 型实数占用 8 个存储单元,即 8 个字节。计算机系统将为保存一个数据分配一个固定大小的存储空间,该空间的大小,即所含的存储单元数等于数据所属类型的长度。根据一个数据的第一个存储单元的地址(称为该数据的首地址,简称该数据的地址)和该数据的类型就可以访问这个数据。所以一个指针通常是一个数据的地址,以一个指针值为存储空间首地址的数据称为指针所指向的数据,称这个指针为指向该数据的指针。

假定 x 是一个 int 型变量,它的值为 100,系统为它分配的存储地址为 p,则 p 所指向的数据为 x, p 为指向 x 的指针。当我们访问 x 时,实际上是访问 p 所指向的 4 个字节内保存的整数 100,这将由系统自动完成,无须用户过问。指针 p 和它所指向的数据 x 可形象地用图 5-1(a)表示出来,其中矩形框表示为 x 分配的存储空间,带箭头的线段表示“指向”。因为指向 x 的指针 p 也需要存储起来,即占用一个存储空间,所以 p 指向 x,也可以用图 5-1(b)表示出来。

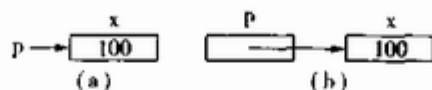


图 5-1 指针与指向数据之间的图形表示

在 C++ 中指针也是一种数据类型,存储每个指针占用四个字节的存储空间,也就是说,指针类型的长度为 4,它同整数型(int)、枚举型(enum)和单精度型(float)具有相同的长度。

按理说,占用 4 个字节的任何一个二进制整数都是一个指针常数,它表示内存中一个相应的内存单元,但它们只能提供给操作系统和编译系统使用,用户定义的对象必须由系统分配存储空间,用户不能直接使用地址常数为对象分配存储空间。因此,所有指针常数都不允许用户使用,但有一个指针常数 0 例外,它允许被用户使用,其含义已经不是指针编址为 0 的存储单元,而表示指针为空,即不指向内存中任何存储单元。在 iostream.h 头文件中,符号常量 NULL 被定义为常数 0。

在 C++ 程序中通常使用的是指针变量,用它存储一个数据(对象)的地址,当然这个地址是在定义对象时由系统自动分配的。若要通过指针(变量)访问它所指向的数据,必须同

时知道它所指向的数据的类型,因为类型不同,一次存取的存储空间的大小就可能不同,对存取内容的解释也可能不同。因此,要知道一个指针所指向数据的类型,必须把一个指针定义为指向某种数据类型的指针。如把一个指针变量定义为指向 `int` 类型的指针,当通过该指针存取它所指向的数据时将是一个整数,若把一个指针变量定义为指向 `double` 类型的指针时,存取它所指向的数据将是一个双精度数。

一个指针所指向数据的类型,有时被称之为该指针的类型。如一个指针所指向的数据为 `int` 型,则称该指针为 `int` 型,当然更确切地应为 `int *` 型,又如一个指针所指向的数据为 `char` 型,则称该指针为 `char` 型,或确切地称为 `char *` 型。

## 5.2 指针变量

### 1. 定义格式

<类型关键字> \* <指针变量名> [= <指针表达式>], ...;

定义指针变量同定义普通变量一样,都需要给出类型名(即类型关键字)和变量名,同时可以有选择地给出初值表达式,用于给变量赋初值,当然,初值表达式的类型应与赋初值号左边的被定义变量的类型相一致。

定义指针变量与定义普通变量有不同之处:它要在指针变量名前加上星号 \* 字符,表示后跟的为指针变量,而不是普通变量。此星号字符的前后可以不留空格,也可以带有任意多个空格。

最简单的指针表达式是在一个变量名前面加上取地址操作符 &。如 `x` 是一个变量,则 `&x` 的值为 `x` 所在存储空间的首地址。把一个变量的地址赋给一个指针变量后,通过该指针变量能够存取它所指向变量的值。

另外,定义指针变量的类型关键字,除了可以使用定义普通变量的类型关键字外,还可以使用指针类型关键字和 `void` 关键字。指针类型关键字由普通类型关键字后加星号 \* 所组成,如 `int *` 为 `int` 指针类型关键字。`void` 是一个特殊的类型关键字,它只能用来定义指针变量,表示该指针变量无类型,或者说只指向一个存储单元,不指向任何具体的数据类型。

### 2. 格式举例

- (1) `int * p;`
- (2) `int a = 10, * pa = &a;`
- (3) `char c = 'a', * cp = &c;`
- (4) `char * hp1 = "abc", * hp2 = hp1;`
- (5) `void * p1 = 0, * p2 = cp;`
- (6) `double * dp[5], * q = dp[0];`
- (7) `int * ip[10] = {0};`
- (8) `char * rp[3] = {"front", "middle", "rear"};`
- (9) `int n = 20, * np = &n, * * pp = &np;`

第一条语句定义  $p$  为一个整型指针变量,即  $p$  的类型为  $\text{int}^*$ , $p$  占用 4 个字节的存储空间,用来存储一个整数变量(即类型为  $\text{int}$  的变量)的地址。

第二条语句定义一个整型变量  $a$  和一个整型指针变量  $pa$ , $a$  被初始化为 10, $pa$  被初始化为  $a$  的地址,使  $pa$  指向变量  $a$ ,通过  $pa$  可存取  $a$  的值。

第三条语句定义了一个字符变量  $c$  和一个字符指针变量  $cp$ , $c$  被初始化为字符 'a', $cp$  被初始化为  $c$  的地址,使  $cp$  指向变量  $c$ ,通过  $cp$  可以取出  $c$  的值 'a' 和向  $c$  存入一个新字符。

第四条语句定义了两个字符指针变量  $hp1$  和  $hp2$ , $hp1$  被初始化为字符串常量 "abc" 所在存储空间的首地址。因为在 C++ 语言中规定,一个字符串可以初始化或直接赋值给一个字符指针变量,但这种赋值不是赋字符串本身,而是赋存储该字符串的地址。对  $hp1$  进行初始化后,它就指向了字符串 "abc",通过  $hp1$  可以访问到这个字符串。此语句对  $hp2$  也进行了初始化,使它等于  $hp1$  的值,这样  $hp2$  也指向了字符串 "abc"。

第五条语句定义了两个无类型指针变量  $p1$  和  $p2$ ,并对  $p2$  初始化为  $cp$  的值,结合第三条语句,可知  $p2$  的值为字符变量  $c$  的存储单元的地址。

在 C++ 语言中,利用变量定义语句不仅可以定义指针变量,也可以定义指针数组,就像定义普通变量和数组一样。第六条语句定义了一个  $\text{double}$  指针数组  $dp$ ,该数组包含有 5 个元素  $dp[0] \sim dp[4]$ ,每个元素为一个  $\text{double}$  指针变量,用来保存一个双精度变量的地址。因  $dp$  数组的长度为 5,每个元素的类型(即  $\text{double}^*$  类型)长度为 4,所以整个数组占用 20 个字节的存储空间,当然该存储空间的首地址被保存在数组名  $dp$  中。该语句同时定义了一个  $\text{double}$  指针变量  $q$ ,并对它初始化为  $dp[0]$  的值,因为  $dp[0]$  为  $\text{double}$  指针型,它同  $q$  具有相同的类型,所以可以把它赋给  $q$ 。反过来,一个指针变量的值也可以赋给一个同类型的指针元素。

第七条语句定义了一个整型指针数组  $ip[10]$ ,该数组中的每个元素都是一个  $\text{int}^*$  型变量,各自用于保存一个整数存储空间的地址。该语句对  $ip$  数组进行了初始化,使得每个元素的值为 0,即空指针。

第八条语句定义了一个字符指针数组  $rp[3]$ ,它的每个元素都是一个字符指针变量,并且分别被初始化为相应字符串常量的地址,使得  $rp[0]$  指向 "front" 字符串, $rp[1]$  指向 "middle" 字符串, $rp[2]$  指向 "rear" 字符串,如图 5-2 所示。

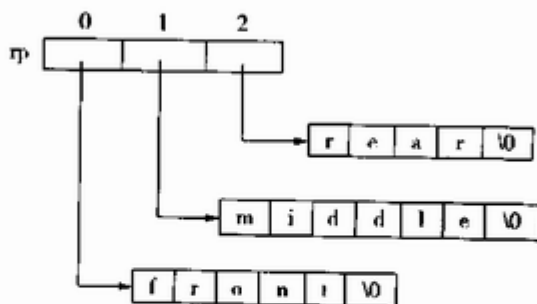


图 5-2 字符指针指向字符串数据的图形表示

第九条语句定义了一个整型变量  $n$  并初始化为 20,又定义了一个整型指针变量  $np$ ,并被初始化为  $n$  的地址,使之指向  $n$ ,还定义了一个整型二级指针变量  $pp$ ,它指向一个整型指

针变量,即指向类型为 `int *` 的变量。因为 `np` 就是类型为 `int *` 的变量,所以可以用它对 `pp` 进行初始化。注意:`np` 和 `pp` 具有不同的指针类型,前者指向的数据是一个整数,后者指向的数据是一个整数指针,即它存储的是一个整数指针的存储空间的首地址。`n`,`np` 和 `pp` 之间的关系如图 5-3 所示。

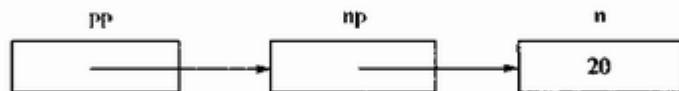


图 5-3 指针与指向数据之间的关系

假定系统编译后运行时给 `n`,`np` 和 `pp` 分别分配的相应四个字节的存储空间的首地址依次为十六进制的 `0xFDF4`,`0xFDF0` 和 `0xFDEC`,则 `np` 的值应为 `0xFDF4`,`pp` 的值应为 `0xFDF0`。当程序要求访问 `n`,`np` 和 `pp` 时,就是分别从 `0xFDF4~0xFDF7`, `0xFDF0~0xFDF3` 和 `0xFDEC~0xFDEF` 的存储空间中存取内容。

**注意:**将 `&n` 的值赋给 `pp` 指针是错误的,因为 `&n` 的值具有的类型为 `int *`,而 `pp` 指针的类型为 `int **`。

### 3. 几点说明

(1) C++ 语言中的每一种数据类型同其后面的一个星号(`*`)结合起来就构成了一种新的数据类型,即指针数据类型,但每个星号只对它后面的一个变量名起作用,而对逗号之后的下一个变量无效。如:

```
int *p1, p2;
```

`p1` 被定义为 `int *` 类型,即指向整型的指针类型,而 `p2` 被定义为 `int` 类型,若要把 `p2` 也说明为 `int *` 类型,则必须在 `p2` 之前加上星号,变为如下形式的定义:

```
int *p1, *p2;
```

(2) 指针数据类型也同样是一种数据类型,它同后面的一个星号的结合又产生出一种新的指针数据类型,这个新的指针数据类型可以继续同星号结合产生新类型,以此类推。如:

```
int *p1, **p2, ***p3;
```

`p1`,`p2` 和 `p3` 的类型依次为 `int *`,`int **` 和 `int ***`,它们都是指针类型,并且最基本的类型都为 `int`,但指针的级别不同,`p1` 称为一级整型指针,`p2` 和 `p3` 分别称为二级和三级整型指针。一级指针指向普通(即非指针)数据对象,二级指针指向一级指针的数据对象,三级指针指向二级指针的数据对象,等等。

(3) 给一个指针变量的赋值必须是同类型的指针表达式的值,但对 `void *` 类型的指针变量的赋值可以为任何类型的指针表达式的值,反过来则不允许。

- ① `char * p = "string", ** q = &p;`
- ② `void * v1 = p, * v2 = (void *)p;`
- ③ `char * cp = v2;`

在第一条语句中,定义指针 `p` 为 `char*` 类型,并把赋初值号右边的“string”的值(即该字符串的首地址)赋给 `p`,由于字符串的首地址就是存储第一个字符的存储单元的地址,它的类型为 `char*`,所以它与 `p` 具有相同的类型。该语句定义的指针 `q` 为 `char**` 类型,而用于对它赋初值的表达式 `&p` 是具有 `char*` 类型的 `p` 的地址,所以该表达式的类型为 `char*`,把这个值赋给 `q` 是合法的。这条语句也可以改写为:

```
char *p;p="string";char **q;q=&p;
```

上述第二条语句定义 `v1` 为 `void*` 类型,并给它赋初值为字符指针 `p` 的值,虽然类型不同,但是允许的,系统将自动把这个值转换为左边的 `void*` 类型,然后再实现赋值。若反过来把一个 `void*` 类型的值赋给除该类型之外的其他任何指针类型则是不允许的。该语句同时定义了无类型指针变量 `v2`,并把具有 `char*` 类型的指针变量 `p` 的值经强制转换为 `void*` 类型后赋给 `v2`,此时 `v2` 和 `v1` 一样,其值均为字符串“string”的首地址。

上述第三条语句是错误的,因为它试图把一个无类型的指针赋给一个有类型的指针变量。

(4) 若需要把一个指针表达式的值赋给一个与之不同的指针类型的变量时,应把这个值强制转换为被赋值变量所具有的指针类型,当然在转换前后,只是类型发生了变化,其具体的地址值(即一个十六进制的整数代码)不变。如:

```
char *cp;
int a[10];
cp=(char*)&a[0];
```

在这里 `cp` 为 `char*` 类型的指针变量,而 `&a[0]` 为 `int*` 类型的表达式,要把这个表达式的值赋给 `cp` 必须把它强制转换为 `char*` 类型。关于这种转换的用途,将在最后的文件一章中见到。

(5) 在 `cout` 语句中除了可以输出普通表达式的值之外,也同样可以输出指针表达式的值。但要注意:当指针表达式的值为一个 `char*` 类型的指针时,不是输出这个指针值,而是输出这个值所指向(即以这个值为首地址)的字符串。如:

```
int a=30,*ap=&a;
char *cp;
cp="output";
cout << ap << ' ' << cp << endl;
cout << (void*)cp << endl;
```

当执行第一条 `cout` 语句时,输出的 `ap` 的值就是整型变量 `a` 的存储地址,输出 `cp` 数据项时不是输出它的指针值,而是输出它指向的字符串“output”。执行第二条 `cout` 语句时,由于把字符指针 `cp` 的值转换成了一个 `void*` 类型的值,所以此处输出的是真正的 `cp` 值,即存储字符串常量“output”的首地址。该程序段的执行结果为:

```
0x0065FDF4 output
0x004260B4
```

上面输出的地址值是在一种机器上运行得到的,对于不同的机器可能会得到不同的地址值。



在 4.4 节中已经讲过,当把保存字符串的数组名或一个字符串常量作为输出项时,输出的是数组中的字符串或所给的字符串本身,这是因为字符数组名保存的是该数组的首地址,即第一个字符的地址,它的类型为 `char *`,字符串常量的值是存储该字符串的首地址,它也具有 `char *` 类型。

(6) 在 `cin` 输入语句中,除了能够使用普通变量输入数据外,只能够再使用字符指针变量输入字符串数据,不能够使用其他任何指针变量输入数据。但要注意:用于输入的字符指针变量必须指向了一个字符数组空间,并且该数组的长度要大于待输入的字符串的长度。如:

```
char a[30];
char *p = a;
cin >> p;
cout << p << ' ' << a << endl;
```

该程序段首先定义了一个字符数组 `a[30]`,接着定义了一个字符指针变量 `p`,并给它赋初值为 `a`,即数组 `a` 的首地址,然后等待用户从键盘上输入一个字符串,输入结束并按下回车键后,将把它存入到 `p` 指针所指向的存储空间中,即数组 `a` 的空间中。当然输入的字符串的长度要小于 30,此程序段的最后一行语句分别输出 `p` 和 `a` 所指向的字符串,因为 `p` 和 `a` 的指针相同,所以将输出同一个字符串,即在数组 `a` 中保存的字符串。

(7) 在一条变量定义语句的前面加上 `const` 保留字,将使得所定义的普通变量为常量,即除了在定义时赋初值外,其后只允许读取它的值而禁止对它的修改,同时使所定义的指针变量为常量指针(即指向常量的指针),对它所指向的对象只能被读取,而不允许被修改,这个指针变量的值可以被修改。如:

- ① `const int a[3] = { -1, 0, 1 };`
- ② `int n = 20, m = 30;`
- ③ `const int *p = &n;`
- ④ `p = &m;`
- ⑤ `const char b[10] = "computer";`
- ⑥ `char c[12] = "television";`
- ⑦ `const char *pp = b;`
- ⑧ `pp = "pointer";`

第一条语句定义 `a[3]` 为整型常量数组,其三个元素依次被初始化为 -1, 0 和 1,该数组被初始化后不允许被修改,而只能够从中读取每个元素的值。

第二条语句定义了 `n` 和 `m` 这两个整数变量,前者被初始化为 20,后者被初始化为 30。

第三条语句定义了一个整型常量指针 `p`,并被初始化为整数变量 `n` 的地址,使 `p` 初始指向 `n`。由于 `p` 是一个常量指针,所以不允许以后修改 `p` 所指向的对象,但允许修改 `p` 的值,使之指向另一个对象。以后同样也不允许通过 `p` 修改当前所指向的另一个对象。

第四条语句使常量指针 `p` 指向了另一个整数对象 `m`。

第五条语句定义 `b[10]` 为字符型常量数组并被初始化,该数组中的每个元素只允许被读取,不允许被修改。

第六条语句定义了一个字符数组 `c[12]` 并被初始化,该数组的内容既可以被读取也可

以被修改。

第七条语句定义了一个字符型常量指针 pp,并使之指向 b 数组中保存的字符串,由于 pp 是常量指针,所以不允许修改 pp 所指向的对象(当前为数组 b)。若把 pp 初始化为 c 的值,使 pp 指向数组 c 中的字符串,则以后也不允许修改 pp 所指向的对象(当前为数组 c)。

第八条语句把一个字符串常量"pointer"的首地址赋给了字符指针 pp,这是允许的,因为虽然不允许修改常量指针所指向的对象,但允许修改它本身的值,使之指向另一个对象。

另外,若把 const 保留字放在变量定义语句中的星号 \* 与变量名之间,则定义的指针变量为一个指针常量,即不允许修改该指针变量的值。如:

① char \* const cp = "const";

② cp = "variable";

第一条语句定义 cp 为一个字符型指针常量,它指向字符串"const",以后不允许修改 cp 的值,使它指向其他存储位置。第二条语句是非法的,因为它试图修改指针常量 cp 的值。

这里顺便指出,一个字符串常量是被存储在内存中常量数据区内。无论把它的地址赋给任何字符指针变量,都不允许通过这个指针修改所指向的字符串常量。

(8) 使用 typedef 语句也同样能够把一种指针类型定义为新的名字。如:

① typedef int \* inPointer;

② typedef char \* chPointer;

③ typedef char \* AA[10];

第一条语句定义标识符 inPointer 为 int \* 类型的别名,以后利用这个标识符可以定义出类型为 int \* 的对象。同理,利用第二条语句定义的 chPointer 标识符能够定义出类型为 char \* 的变量,利用第三条语句定义的 AA 标识符能够定义出类型为 char \* [10]的变量,该类型为含有 10 个字符指针的数组类型。如:

(1) inPoint a;

(2) chPointer b = 0;

(3) AA c = {"zero", "one"};

其中 a 为 int \* 类型, b 为 char \* 类型, c 为 char \* [10]类型, b 被初始化为 NULL, c 中的元素 c[0]和 c[1]分别被初始化为"zero"和"one"的地址,其余所有元素被初始化为 NULL。

## 5.3 指针运算

对指针也可以进行一些运算,如赋值、取地址、间接访问、比较、增 1、减 1 等。

### 1. 赋值(=)

指针之间也能够赋值,它是把赋值号右边指针表达式的值赋给左边的指针对象,该指针对象必须是一个左值,并且赋值号两边的指针类型必须相同。但有一点例外,那就是允许把任一类型的指针赋给 void \* 类型的指针对象。如:

```
char ch = 'd', * cp;  
cp = &ch; // 把 ch 的地址赋给 cp
```

## 2. 取地址(&)

取地址操作符是单目操作符,它被用在一个指针对象的前面,运算结果是该指针对象的地址。如:

```
int x = 30, *xp = &x;
cout << x << ' ' << xp << endl; // 输出 x 的值和 x 的地址
```

取地址操作符的操作对象必须是一个左值,其操作结果是一个右值(即非左值)。

## 3. 间接访问(\*)

间接访问操作符也是一个单目操作符,它的后面是一个指针操作数,操作结果是该指针所指的对象。如:

```
int x = 10, y = 20;
int *xp = &x, *yp = &y;
cout << *xp << ' ' << *yp << endl;
int z = *xp + *yp;
*xp += 5;
cout << *xp << ' ' << *yp << ' ' << z << endl;
```

此程序段中第一条语句定义了两个整型变量  $x$  和  $y$  分别赋初值为 10 和 20;第二条语句定义两个整型指针变量  $xp$  和  $yp$ ,并分别赋初值为  $x$  和  $y$  的地址;第三条语句依次输出  $xp$  和  $yp$  所指对象(对应为  $x$  和  $y$ )的值;第四条语句定义了一个整型变量  $z$ ,使它初始化为表达式  $*xp + *yp$  的值,因为  $*xp$  为  $xp$  所指的对象  $x$ , $*yp$  为  $yp$  所指的对象  $y$ ,所以该表达式等同于  $x + y$ ,其值为 30;第五条语句把  $*xp$  的值加上 5 后再赋给  $*xp$ ,因  $*xp$  指的是  $x$ ,所以相当于执行  $x += 5$  的操作;第六条语句依次输出  $*xp$ ,  $*yp$  和  $z$  的值,该程序段的运行结果为:

```
10 20
15 20 30
```

又如:

```
char c1 = 'a', c2 = 'b', c;
cout << c1 << ' ' << c2 << endl;
char *cp = &c1, *cq = &c2;
c = *cp; *cp = *cq; *cq = c;
cout << c1 << ' ' << c2 << endl; // 相当于 cout << *cp << ' ' << *cq << endl;
```

此程序段的第一行定义了字符变量  $c1$ ,  $c2$  和  $c$ ,并对  $c1$  和  $c2$  分别初始化为 'a' 和 'b';第二行输出  $c1$  和  $c2$  的值;第三行定义了字符指针变量  $cp$  和  $cq$ ,并分别初始化为  $c1$  和  $c2$  的地址;第四行三条语句实现  $c1$  和  $c2$  值的交换,相当于执行“ $c = c1$ ;  $c1 = c2$ ;  $c2 = c$ ;”这三条语句;第五行又重新输出  $c1$  和  $c2$  的值。该程序段的运行结果为:

```
a b
b a
```

间接访问操作符的一个指针操作数,既可以是左值,又可以是右值,其操作结果是该指针所指向的对象,它是一个左值。

间接访问操作符是一个星号,它与定义指针变量所使用的星号相同,同时星号也是算术运算中的双目运算符乘号,用户应根据星号出现的位置加以区分,判断其作用。

假定  $x$  是一个变量,则  $*\&x$  的结果仍为  $x$ ,这是因为,按照  $*$  和  $\&$  的运算规则,它们属于同一级运算,并且其结合性是从右向左,所以先进行  $\&$  运算取出  $x$  的地址,再进行  $*$  运算访问该地址所指向的对象  $x$ ,因此整个运算结果仍为  $x$ ;同样,若  $p$  是一个指针对象,则  $*p$  的值仍为  $p$  的值,因为应先进行  $*$  运算得到  $p$  所指的對象,接着进行  $\&$  运算得到该对象的地址,该地址就是  $p$  的值。

#### 4. 增 1(++)和减 1(--)

增 1 和减 1 操作符同样适应于指针类型,使指针值增加或减少所指数据类型的长度值。设  $p$  是一个指向 A 类型的指针变量,则  $p++$  表示先得到  $p$  的值,然后  $p$  增 1,实际上  $p$  增加了 A 类型的长度值,使  $p$  指向了原数据的后面一个数据。如:

```
int a[4] = {10,25,36,48};
int *p = a;
cout << *p << ' ';
p++;
cout << *p++ << ' ';
cout << *++p << endl;
```

该程序段的第一行定义了一个整型数组  $a[4]$  并被初始化;第二行定义了一个整型指针  $p$  并使之指向数组  $a$  中的第一个元素  $a[0]$ ;第三行输出  $p$  所指向的对象  $a[0]$  的值;第四行使  $p$  增加 1,使之指向数组  $a$  中的第二个元素  $a[1]$ , $p$  所保存的地址值实际上是增加了  $\text{int}$  型的长度 4;第五行输出表达式  $*p++$  的值和一个空格,计算表达式  $*p++$  的值时,因  $++$  和  $*$  是同一级单目运算,并且按从右向左结合,所以先计算  $++$  后计算  $*$ ,计算  $p++$  得到  $p$  的值,接着计算  $*$  得到  $p$  所指向的对象  $a[1]$ ,同时在计算  $p++$  返回  $p$  的值之后又使  $p$  增 1,这时  $p$  指向了下一个元素  $a[2]$ ;第六行输出表达式  $*++p$  的值和一个回车,计算  $*++p$  时,应先计算  $++$  后计算  $*$ ,使输出的是  $a[3]$  的值。该程序段的运行结果为:

10 25 48

对于表达式  $*p++$  若写成  $(*p)++$ ,则将首先访问  $p$  所指向的对象,然后使这个对象的值增 1,而指针  $p$  的值将不变。

又如:

```
char b[10] = "abcdef";
char *p = b;
cout << *p++ << ' ';
p++;p++;
cout << *p-- << ' ';
cout << *--p << endl;
```

该程序段的第一行定义了一个字符数组  $b[10]$  并被初始化为 "abcdef";第二行定义了一个字符指针  $p$  并使之指向数组  $b$  中的第一个元素  $b[0]$ ;第三行输出  $p$  所指向的元素  $b[0]$  的值并修改  $p$  使之指向下一个元素  $b[1]$ ;第四行两次使  $p$  增 1,此时  $p$  就指向了元素  $b[3]$ ;第

五行输出元素 b[3] 的值并使 p 指向前一个元素 b[2]; 第六行使 p 指向前一个元素 b[1] 并输出该元素的值。该程序段的运行结果为:

```
a d b
```

### 5. 加(+)和减(-)

一个指针可以加上或减去一个整数(假定为 n), 得到的值将是该指针向后或向前第 n 个数据的地址。如:

```
char a[10] = "ABCDEF";
int b[6] = {1, 2, 3, 4, 5, 6};
char *p1 = a, *p2;
int *q1 = b, *q2;
p2 = p1 + 4; q2 = q1 + 2;
cout << *p1 << ' ' << *p2 << ' ' << * (p2 - 1) << endl;
cout << *q1 << ' ' << *q2 << ' ' << * (q2 + 3) << endl;
```

该程序段的运行结果为:

```
A E D
1 3 6
```

一个指针也可以减去另一个指针, 其值为它们之间的数据个数。若被减数较大则得到正值, 否则为负值。如:

```
double a[10] = {0};
double *p1 = a, *p2 = p1 + 8;
p1 ++; -- p2;
cout << p2 - p1 << ' ' << p1 - p2 << endl;
```

该程序段的运行结果为:

```
6 - 6
```

### 6. 加赋值(+=)和减赋值(-=)

这两种操作是加、减操作和赋值操作的复合。如:

```
int a[5] = {2, 4, 6, 8, 10};
int *p1 = a + 1, *p2 = a + 4;
cout << *p1 << ' ' << *p2 << endl;
p1 += 2; p2 -= 3;
cout << *p1 << ' ' << *p2 << endl;
```

该程序段的运行结果为:

```
4 10
8 4
```

### 7. 比较(==, !=, <, <=, >, >=)

因为指针是一个地址, 地址也有大小, 即后面数据的地址大于前面数据的地址, 所以两

个指针可以比较大小。设  $p$  和  $q$  是两个同类型的指针,则当  $p$  大于  $q$  时,关系式  $p > q, p \geq q$  和  $p != q$  的值为 `true`,而关系式  $p < q, p \leq q$  和  $p == q$  的值为 `false`;若  $p$  的值与  $q$  的值相同,则关系式  $p == q, p \leq q$  和  $p \geq q$  成立,其值为真,而关系式  $p != q, p < q$  和  $p > q$  不成立,其值为假;当  $p$  小于  $q$  时,也可进行类似的分析。

单个指针也可以同其他任何对象一样,作为一个逻辑值使用,当它的值不为空时则为逻辑值 `true`,否则为逻辑值假。判断一个指针  $p$  是否为空,若为空则返回 `true`,否则返回 `false`,该条件可表示为 `!p` 或 `p == 0` 或 `p == NULL`。若要判断一个指针  $p$  是否为空,若不为空则返回 `true`,否则返回 `false`。该条件可表示为 `p` 或 `p != 0` 或 `p != NULL`。

## 5.4 指针与数组

### 5.4.1 指针与一维数组

一维数组是一个组合数据对象,该对象由一定数目的元素对象所组成,整个数组占用内存中一块连续的存储空间,该空间的大小等于所含元素的个数乘以元素类型的长度。

假定  $a[n]$  是一个元素类型为 `int` 的一维数组,该数组占用  $4n$  个字节的存储空间,其存储空间的首地址,亦即第一个元素  $a[0]$  的存储地址,通过使用数组名  $a$  可以访问到这个地址。由数组名可以得到数组中第一个元素的地址,或者说数组名是指向第一个元素的指针,并且数组名不可能指向其他位置,所以可把数组名看做为一个指针常量,为指向元素类型的指针类型。如对于数组  $a$ ,它的值为  $a[0]$  的地址,这个地址的类型为 `int*`。

对于上面假定的一维数组  $a[n]$ ,下标为  $i$  的元素  $a[i]$  的存储地址为  $a+i$ ,因为  $a[i]$  是  $a$  所指向的  $a[0]$  元素之后的第  $i$  个元素。根据指向  $a[i]$  的指针  $a+i$ ,可利用间接访问操作符 `*` 访问  $a[i]$  元素,其访问表达式为 `*(a+i)`。所以访问数组元素有两种方式,一种是下标方式,另一种是指针方式。利用中括号运算符访问数组元素的方式称为下标方式,利用星号运算符访问数组元素的方式称为指针方式。例如:

```
int a[10], i, s = 0;
for(i = 0; i < 10; i++) cin >> a[i];
for(i = 0; i < 10; i++) {
    s += a[i]; cout << a[i] << ' ';
}
cout << endl << s << endl;
```

该程序段向数组  $a[10]$  输入 10 个整数,接着把每一个元素值累加到  $s$  上并输出每个元素值,最后输出累加和  $s$  的值。该程序段可以改写为:

```
int a[10], i, s = 0;
for(i = 0; i < 10; i++) cin >> *(a+i);
for(i = 0; i < 10; i++) {
    s += *(a+i);
    cout << *(a+i) << ' ';
}
```

```
cout << endl << s << endl;
```

由于数组名是指针常量,其值不能被改变(也不应该被改变,若改变了就无法再找到该数组),所以不能够对数组名施加增1或减1运算。但若用一个指针变量指向一个数组,则可改变这个指针变量的值,从而使它指向数组中任何一个元素。据此可将上述程序段改写如下:

```
int a[10], i, s = 0;
int *p = a; // p 指向数组 a 的第一个元素 a[0]
for(i = 0; i < 10; i++) cin >> *p++;
p = a; // 使 p 重新指向数组 a 的开始位置
for(i = 0; i < 10; i++) {
    s += *p;
    cout << *p++ << ' ';
}
cout << endl << s << endl;
```

使用指针变量指向数组后,同样有下标和指针两种访问数组元素的方式。若把上述程序段改写为下标访问方式则为:

```
int a[10], i, s = 0;
int *p = a; // p 指向数组 a 的第一个元素 a[0]
for(i = 0; i < 10; i++) cin >> p[i];
p = a; // 使 p 重新指向数组 a 的开始位置
for(i = 0; i < 10; i++) {
    s += p[i];
    cout << p[i] << ' ';
}
cout << endl << s << endl;
```

对于一个存储字符串的数组,其数组名就是指向其字符串的指针,因为它的值为字符串中第一个字符的存储地址。实际上,指向一个字符串中任一字符位置的指针都是一个指向字符串的指针,该字符串从所指位置开始到末尾空字符为止,它是整个字符串的一个子串。例如:

```
char s1[] = "StringPointer";
char *sp = s1; // s1 的值为 char* 类型
cout << sp << endl;
cout << s1 + 6 << endl;
char s2[10];
for(int i = 0; i < 6; i++) s2[i] = sp[i];
s2[i] = 0;
cout << s2 << ' ' << &s1[6] << endl; // &s1[6] 等同于 s1 + 6
```

该程序段的运行结果为:

```
StringPointer
Pointer
String Pointer
```

对于每个字符串常量,从任一字符开始也都是一个字符串,它是整个字符串的一个尾部

子串。如：

```
char* s1="AddSubtract";
char* s2=s1+3;
cout << s1 << ' ' << s2 << endl;
char s3[10];
strcpy(s3,s2);
s3[3]='\0';
cout << s3 << ' ' << s3+4 << endl;
```

该程序段的运行结果为：

```
AddSubtract Subtract
Sub ruct
```

**注意：**数组名是一个数组类型的变量，类型长度为该数组所占用的存储空间的大小。如在上述程序段中的 s3 就是一个数组变量，其类型为 char [10]，即含有 10 个字符元素的数组类型，类型长度为 10。但当只访问数组名时，它返回的是该数组的第一个元素的地址，编程者得到这个地址后可以访问到该数组中的每个元素。而单纯的指针名是一个指针变量，它的类型长度为 4，即存储一个指针所占有的字节数。如在上述的程序段中，s1 和 s2 都是一个字符指针变量，各占用内存中 4 个字节的存储空间。

#### 5.4.2 指针与二维数组

一个二维数组可首先看做为仅带有第一维下标的一维数组，该数组中的每一个元素又都是一个仅带有第二维下标的一维数组。这样把二维数组分解为两层一维数组后，就可以按照讨论一维数组的方法来讨论二维数组了。例如：

```
int a[M][N]; // M 和 N 为已定义的整型常量
```

此语句定义了一个二维数组 a，第一维大小（即行数）为 M，第二维大小（即列数）为 N，按照第一维得到的一维数组为 a[M]，所含元素依次为 a[0]，a[1]，…，a[M-1]，其中每个元素 a[i] (0 ≤ i ≤ M-1) 又都是一个具有第二维大小的一维数组，所含元素依次为 a[i][0]，a[i][1]，…，a[i][N-1]，a[i] 就是该一维数组的数组名。因为一个数组的数组名就是指向该数组第一个元素的指针，所以 a[i] 就是指向二维数组 a 中行下标为 i 的元素类型为 int 的一维数组的指针，即 a[i] 的值为 a[i][0] 元素的地址，类型为 int\*。同理，二维数组名 a 是指向第一个元素 a[0] 的指针，由于 a[0] 表示具有 N 个 int 型元素的一维数组，即 a[0] 的类型为 int [N]，所以 a 的值为具有 int(\*)[N] 类型的指针。

由于二维数组名 a 的值为 int(\*)[N] 类型，该值增 1 就使指针后移 4×N 个字节，所以 a+i 指向数组 a 的行下标为 i 的一维数组的开始位置，即 a[i][0] 元素的位置。

对于二维数组 a 中的一维元素 a[i]，其指针访问方式为 \*(a+i)，所以二维数组 a[M][N] 中任一元素 a[i][j] 可等价表示为：

$(*(a+i))[j]$  或  $*(*(a+i)+j)$  或  $*(a[i]+j)$

上式中所加的圆括号确保间接访问操作优先于下标运算符，若不加圆括号，按照 C++ 运算规则，下标运算符优先于间接访问运算符。



在二维数组  $a[M][N]$  中,  $a$ ,  $a[0]$  和  $\&a[0][0]$  的地址值都相同, 但类型不同,  $a$  的值为  $\text{int} (*)[N]$  类型, 而  $a[0]$  和  $\&a[0][0]$  的值均为  $\text{int} *$  类型。  $a+1$  则比  $a$  增加  $4 \times N$  个字节, 而  $a[0]+1$  (或  $\&a[0][0]+1$ ) 则比  $a[0]$  (或  $\&a[0][0]$ ) 只增加 4 个字节。

若把一个指针定义为指向具有  $N$  个元素的一维数组的类型, 并用一个具有列数为  $N$  的二维数组的数组名进行初始化, 则该指针就指向了这个二维数组。通过指向二维数组的指针, 同样可以访问该二维数组元素。例如:

```
int a[3][4] = {{2,4,6,8},{3,6,9,12},{4,8,12,16}};
int (*p)[4] = a; // p 与 a 的值具有相同的指针类型, 均为 int(*)[4] 类型
int i, j;
for(i=0; i<3; i++) {
    for(j=0; j<4; j++)
        cout << setw(5) << p[i][j];
    // 采用下标方式访问 p 所指向的二维数组
    cout << endl;
}
```

该程序段的运行结果为:

```
2   4   6   8
3   6   9  12
4   8  12  16
```

上面的 for 双重循环也可以改写为:

```
for(i=0; i<3; i++) {
    int *q = p[i]; // 或使用 *p++ 或使用 *(p+i)
    for(j=0; j<4; j++)
        cout << setw(5) << *q++; // 采用指针访问方式
    cout << endl;
}
```

还可以改写如下:

```
int *q = a[0]; // 或 &a[0][0] 或 (int *)a
for(i=0; i<3; i++) {
    for(j=0; j<4; j++)
        cout << setw(5) << *q++;
    cout << endl;
}
```

## 5.5 引用变量

引用同指针一样, 不是一种单独的数据类型, 它们必须同其他类型组合使用。如  $\text{int} *$  为  $\text{int}$  指针类型, 该类型的变量用来保存整数对象的存储地址, 而  $\text{int}\&$  构成  $\text{int}$  引用类型, 该类型的变量是对它进行初始化的一个对象的别名, 它共享 (或称引用) 初始化对象所具有的存储空间。

定义引用变量同定义指针变量的格式完全相同, 当然要把定义指针的标记符号  $*$  改为

定义引用的标记符号 &。另外,必须对引用变量初始化,这样系统才能够知道它所引用的对象是谁。

引用变量的定义格式:

<类型关键字> & <引用变量名> = <已定义的同类型变量>;

例如:

```
double x = 10;
cout << &x << endl;
double &y = x, z = 20;
cout << x << ' ' << y << ' ' << z << endl;
cout << &x << ' ' << &y << ' ' << &z << endl;
```

该程序段中的第一条语句定义了一个整型变量 x 并被赋初值为 10;第二条语句输出 x 的地址;第三条语句定义了一个整型引用变量 y 并初始化为 x,这样 y 就成为 x 的别名,即 y 将使用 x 所具有的存储空间,该语句同时定义了一个整型变量 z 并被初始化为 20;第四条语句依次输出 x, y 和 z 的值。第五条语句依次输出 x, y 和 z 的地址。该程序段的运行结果为:

```
0x0066PDF0
10 10 20
0x0066PDF0 0x0066PDF0 0x0066FDE4
```

从运行结果可以看出, y 和 x 占用同一存储空间,即系统为 x 分配的存储空间,显示出的 x 和 y 的值相同,即为共用的存储空间中保存的整数 10。

定义引用变量所使用的符号标记与取对象地址运算符相同,即为 &,读者可根据它所出现的场合判明它的用途,当出现在变量定义语句(或函数参数表)中一个被定义的变量之前时,则表示该变量为引用,当出现在其他任何地方时,则表示为取地址运算符。

由于引用变量是使用它所引用的对象的存储空间,所以对它赋值等价于对它所引用的对象的赋值,反之亦然。如:

```
char h = 'a', &r = h;
cout << h << ' ' << r << endl;
r = 'b';
cout << h << ' ' << r << endl;
h = 'c';
cout << h << ' ' << r << endl;
```

该程序段的运行结果为:

```
a a
b b
c c
```

任何一种数据类型同 & 结合都可以构成引用类型,从而定义出引用变量。下面给出定义指针引用变量的例子。

```
int a[5] = {10, 20, 30, 40, 50};
int *p = a;
int &r = p; // r 是 p 的引用, p 和 r 均指向 a[0] 元素
```

```

cout << &p << ' ' << &r << endl;
cout << p << ' ' << r << ' ' << &a[0] << endl; // p 和 r 的值为 a[0] 的地址
cout << *p << ' ' << *r << endl; // *p 和 *r 表示对象 a[0]
p++; r++;
cout << &p << ' ' << &r << endl;
cout << p << ' ' << r << ' ' << &a[2] << endl; // p 和 r 的值为 a[2] 的地址
cout << *p << ' ' << *r << endl; // *p 和 *r 表示对象 a[2]

```

该程序段的运行结果为：

```

0x0065FDE0 0x0065FDE0
0x0065FDE4 0x0065FDE4 0x0065FDE4
10 10
0x0065FDE0 0x0065FDE0
0x0065FDEC 0x0065FDEC 0x0065FDEC
30 30

```

引用类型主要使用在对函数形参的说明中,使该形参成为传送给它的实参对象的别名,关于这方面的内容将留到下一章讨论。

## 5.6 动态存储分配

在变量定义语句中定义的变量是在程序的编译阶段分配其存储空间的,每个变量对象的存储空间的大小等于所属类型的长度。这种在程序编译阶段对变量进行的存储分配称为静态分配。

在C++语言中,除了静态分配外,还可以进行动态存储分配,即在程序运行阶段实现数据存储空间的分配。静态分配和动态分配将在内存中不同的存储区内进行。

使用 new 操作能够实现动态分配,它是一种单目操作,操作数紧跟其后,该操作数是任一种数据类型。当数据类型不是数组时,还可以初始化由动态分配得到的数据空间。

当程序执行 new 运算时,将首先从内存中相应的存储区内分配一块存储空间,该空间的大小等于 new 运算符后指明的数据类型的长度;然后返回该存储空间的地址,对于数组类型返回的是该空间中存储第一个元素的地址。

若执行 new 操作时,无法得到所需的存储空间,则表明动态分配失败。此时返回空指针,即运算结果为 NULL。

使用 new 操作符的格式为:

```
new <已知数据类型> [( <初值表达式> )]
```

对于非数组类型,中括号内为可选项,对于数组类型,中括号内应给出作为数组长度的表达式。

格式举例:

- (1) new int;
- (2) new int(5);
- (3) new char[10];

- (4) `new int[n];`
- (5) `new double[m+1][N+1];`
- (6) `new char* (&x);`

执行第一条运算时将分配到具有 4 个字节的整数存储空间,并返回该存储空间的地址,即指向该存储空间的指针,该指针的类型为 `int*`。

执行第二条运算时同样分配到具有 4 个字节的整数存储空间,返回该存储空间的地址,并且对该存储空间进行初始化,使之存储一个整数 5。

执行第三条运算时,首先分配到具有 10 个字节的字符数组空间,然后返回该存储空间中存储第一个元素的地址,其返回值类型为 `char*`。

执行第四条运算时,首先分配到能够存储  $n$  个整数的数组空间,然后返回该存储空间首地址,即存储第一个元素的地址,其返回值为 `int*` 类型。

**注意:**当采用 `new` 运算动态分配一维数组空间时,该数组的长度  $n$  既可以为一个常量表达式,又可以为一个变量表达式。而在变量定义语句中定义的数组,其每一维的长度都必须是一个常量表达式,不允许是变量表达式。当只有在程序运行时才能够确定待使用数组的长度时,则只能采用动态分配建立该数组,不能采用变量定义语句定义它。

执行第五条运算时,首先分配  $(m+1) * (N+1)$  个双精度数存储空间,它是一个二维双精度数组空间;然后返回第一个元素的地址。由于对应的一维数组的元素类型为 `double[N+1]`,所以返回值的类型为 `double(*)[N+1]`。

**注意:**当采用 `new` 运算动态分配二维数组空间时,第二维的长度(即列数)必须为常量表达式,第一维的长度(即行数)可以为常量表达式,也可以为变量表达式。在上述第五条运算中, $N$  必须为一个事先定义的整型常量, $m$  可以为常量,也可以为变量。

同理,当采用 `new` 运算动态分配二维以上数组的存储空间时,只有第一维的尺寸是可变的,其余维的尺寸都必须为常量,返回值为一个指向数组的指针,该数组的类型为除上述第一维之外剩下的数组类型。如:

```
new int[2][3][4];
```

返回值的类型为 `int(*)[3][4]`,其值是按第一维考虑的第一个元素的地址。

执行上面第六条运算时,首先动态分配一个 4 字节的用于存储一个字符指针的数据空间,并使这个数据空间初始指向  $x$ ,假定  $x$  是一个 `char` 类型的对象;然后返回这个数据空间的地址。由于该数据空间保存的是字符指针,所以返回值的类型为 `char**`。

采用 `new` 运算能够实现数据存储空间的动态分配,但用户只有把它的返回值保存到一个指针变量后,才能够通过这个指针变量间接地访问这个存储空间(即数据对象)。当然用户所定义的指针变量的类型必须与 `new` 运算返回值的类型相同。如:

- (1) `int * p1 = new int(8);`
- (2) `int * p2 = new int[10];`
- (3) `char (* p3)[12] = new char[6][12];`
- (4) `int ** p4 = new int* (p1);`

第一条语句使 p1 指向动态分配的一个存储整数的空间,该空间被初始化为 8,通过使用 \*p1 可以存取该整数对象。

第二条语句使 p2 指向动态分配的、能够存储 10 个整数的数组空间,通过 p2 指针可以按下标和指针两种方式访问该数组中的任意元素。

第三条语句使 p3 指向一个二维字符数组,该数组为 6 行 12 列,共包含存储 72 个字符的空间,通过 p3 指针可以同使用数组名一样访问该数组。如使用语句:

```
cin >> p3[i];
```

可以从键盘上输入一个字符串到行下标为 i 的一维字符数组 p3[i][12] 中,当然,输入的字符串的长度必须小于等于 11。

第四条语句使 p4 指向一个 int\* 型的指针对象,该对象被初始化为 p1(假定 p1 为 int\* 类型),所以 \*p4 的值就是指向一个整数对象的指针,\* \*p4 的值就是这个整数对象的值。

使用 new 运算符动态分配给用户的存储空间,可以通过使用 delete 运算符重新归还给系统,若没有使用这种运算归还,则只有等到整个程序运行结束才被系统自动回收。

使用 delete 运算的格式为:

```
delete p1; 或 delete []p1;
```

其中 p1 表示指向动态分配的非数组空间的指针,p2 表示指向动态分配的数组空间的指针。如:

```
int *p = new int;
*p = 20; (*p)++;
int x = *p - 5;
cout << *p << ' ' << x << endl;
delete p;
```

该程序段中的第一行得到一个动态分配的整数对象 \*p;第二行首先给 \*p 赋值为 20,然后让它增 1,其值由 20 变为 21;第三行定义整数变量 x,并对它进行初始化,初始表达式为 \*p - 5,值为 16;第四行输出 \*p 和 x 的值;第五行把 p 所指向的动态分配的整数空间归还给系统,注意为 p 静态分配的 4 个字节的指针空间不会被归还,还可以利用 p 指向另一个整数对象。如:

```
int y = 13; p = &y; cout << *p << endl;
```

又使 p 指向了整数对象 y,此时 \*p 就是 y。

下面的程序段含有动态分配和释放数组的操作。

```
int n, i;
cin >> n;
int *a = new int[n];
a[0] = 1;
for(i = 1; i < n; i++)
    a[i] = 2 * a[i - 1] + 1; // 下标访问方式
for(i = 0; i < n; i++)
    cout << * (a + i) << ' '; // 指针访问方式
cout << endl;
```

```
delete []a;
```

该程序段首先定义  $n$  并从键盘上输入一个整数给  $n$ , 接着得到一个动态分配的具有  $n$  个元素的一维数组, 并由  $a$  指针所指向; 再接着给这个数组中的每个元素赋值; 然后依次输出每个元素的值, 程序段最后释放 (即删除) 由  $a$  所指向的动态数组空间。假定从键盘上给  $n$  输入的整数为 10, 则得到的运行结果为:

```
1 3 7 15 31 63 127 255 511 1023
```

下面给出一个使用动态数组空间的完整程序, 供读者参考。

```
#include <iostream.h>
void main()
{
    int n;
    cout << "请输入一个正整数:";
    cin >> n;
    int *a = new int[n];
    cout << "请输入" << n << "个整数:" << endl;
    int i, j;
    for(i = 0; i < n; i++) cin >> a[i];
    for(i = 0; i < n - 1; i++) { // 对 n 个元素进行选择排序
        int k = i;
        for(j = i + 1; j < n; j++)
            if(a[j] < a[k]) k = j;
        int x = a[i]; a[i] = a[k]; a[k] = x;
    }
    for(i = 0; i < n; i++) // 输出排序结果
        cout << a[i] << ' ';
    cout << endl;
    delete []a;
}
```

该程序得到从键盘上输入的  $n$  的整数, 然后再按照从小到大的次序显示出来。

## 习题五

### (一) 填空题

1. 一个指针类型的对象占用内存的\_\_\_\_\_个字节的存储空间。
2. 一个指针指向一个数据对象, 它保存着该数据对象的\_\_\_\_\_, 若数据对象为 `DataType` 类型, 则该指针的类型为\_\_\_\_\_。
3. 若要把一个整型指针  $p$  转换为字符指针, 则采用的强制转换表达式为\_\_\_\_\_。
4. 假定一个数据对象为 `int*` 类型, 则指向该对象的指针的类型为\_\_\_\_\_。
5. 假定  $p$  是一个指向整数对象的指针, 则用\_\_\_\_\_表示该整数对象, 用\_\_\_\_\_表示指针变量  $p$  的地址。
6. 假定  $p$  是一个指针, 则  $*p++$  运算首先访问\_\_\_\_\_, 然后使\_\_\_\_\_的值增 1。

7. 假定  $p$  是一个指针, 则  $(*p)++$  运算首先访问\_\_\_\_\_, 然后使\_\_\_\_\_的值增 1。
8. 假定  $p$  所指对象的值为 25,  $p+1$  所指对象的值为 42, 则  $*p++$  的值为\_\_\_\_\_。
9. 假定  $p$  所指对象的值为 25,  $p+1$  所指对象的值为 42, 则  $*++p$  的值为\_\_\_\_\_。
10. 假定  $p$  所指对象的值为 25,  $p+1$  所指对象的值为 42, 则执行  $(*p)++$  运算后,  $p$  所指对象的值为\_\_\_\_\_。
11. 假定  $p$  所指对象的值为 25,  $p+1$  所指对象的值为 42, 则执行  $*(p++)$  或  $*p++$  运算后,  $p$  所指对象的值为\_\_\_\_\_。
12. 假定  $a$  是一个一维指针数组, 则  $a+i$  所指对象的地址比  $a$  大\_\_\_\_\_字节。
13. 假定  $a$  是一个一维数组, 则  $a[i]$  的指针访问方式为\_\_\_\_\_。
14. 假定  $a$  是一个一维数组, 则  $a[i]$  对应的存储地址(以字节为单位)为\_\_\_\_\_。
15. 一个数组的数组名实际上是指向该数组\_\_\_\_\_元素的指针, 并且在任何时候都不允许\_\_\_\_\_它。
16. 假定指向一维数组  $b[10]$  中元素  $b[4]$  的指针为  $p$ , 则  $p+3$  所指向的元素为\_\_\_\_\_,  $p-2$  所指向的元素为\_\_\_\_\_。
17. 若要定义整型指针  $p$  并初始指向  $x$ , 则所使用的定义语句为\_\_\_\_\_。
18. 若  $p$  指向  $x$ , 则\_\_\_\_\_与  $x$  的表示是等价的。
19. 在一个二维数组  $\text{int } a[m][n]$  中, 包含的一维元素  $a[i]$  的类型为\_\_\_\_\_, 访问  $a[i]$  时返回值的类型为\_\_\_\_\_。
20. 假定一个二维数组为  $c[5][8]$ , 则  $c[3]$  的值为二维元素\_\_\_\_\_的地址,  $c[3]+2$  的值为二维元素\_\_\_\_\_的地址。
21. 假定  $p$  为指向二维数组  $\text{int } d[4][6]$  的指针, 则  $p$  的类型为\_\_\_\_\_。
22. 假定一个二维数组为  $\text{char } f[10][20]$ , 则一维元素  $f[i]$  的类型为\_\_\_\_\_, 访问  $f[i]$  时返回值的类型为\_\_\_\_\_, 其值是元素\_\_\_\_\_的地址。
23. 假定  $a$  是一个二维数组, 则  $a[i][j]$  的指针访问方式为\_\_\_\_\_。
24. 若要把整型变量  $y$  定义为  $x$  的引用, 则所使用的定义语句为\_\_\_\_\_。
25. 若  $y$  是  $x$  的引用, 则对  $y$  的操作就是对\_\_\_\_\_的操作。
26. 若  $y$  是  $x$  的引用, 则  $\&y$  和  $\&x$  的值\_\_\_\_\_, 即为变量\_\_\_\_\_的地址。
27. 执行  $\text{int } *p = \text{new int}$  操作可得到一个动态分配的整型对象\_\_\_\_\_。
28. 执行  $\text{int } *p = \text{new int}[10]$  操作, 使  $p$  指向动态分配的数组中下标为\_\_\_\_\_的元素, 该元素可表示为\_\_\_\_\_或\_\_\_\_\_。
29. 执行  $\text{char } *p = \text{new char}('a')$  操作后,  $p$  所指向的数据对象的值为\_\_\_\_\_。
30. 执行  $\text{new char}[m][n]$  操作时的返回值的类型为\_\_\_\_\_。
31. 执行\_\_\_\_\_操作将释放由  $p$  所指向的动态分配的数据空间。
32. 执行\_\_\_\_\_操作将释放由  $p$  所指向的动态分配的数组空间。

## (二) 写出下列每个程序运行后的输出结果并上机验证

```
1. #include <iostream.h>
   void main()
   {
```

```

char a[15] = "1352460357589";
char *p = a;
int c1, c3, c5;
c1 = c3 = c5 = 0;
while(*p) {
    if(*p == '1') c1++;
    else if(*p == '3') c3++;
    else if(*p == '5') c5++;
    p++;
}
cout << c1 << ' ' << c3 << ' ' << c5 << endl;
}

```

```

2. #include <iomanip.h>
void main() {
    int a[8] = {3, 5, 7, 9, 11, 13, 15, 17};
    int *p = a;
    for(int i = 0; i < 8; i++) {
        cout << setw(5) << *p++;
        if((i+1) % 4 == 0) cout << endl;
    }
}

```

```

3. #include <iostream.h>
void main()
{
    int a[8] = {46, 38, 72, 55, 24, 63, 50, 37};
    int s = 0;
    int *p = a + 3;
    while(p < a + 8) s += *p++;
    cout << s << ' ' << s/5 << ' ' << float(s)/5 << endl;
}

```

```

4. #include <iostream.h>
void main()
{
    int a[8] = {46, 38, 72, 55, 24, 63, 50, 37};
    int max = *a, min = *a;
    for(int *p = a + 1; p < a + 8; p++) {
        if(*p > max) max = *p;
        if(*p < min) min = *p;
    }
    cout << max << ' ' << min << endl;
}

```

```

5. #include <iomanip.h>
void main() {
    int a[5] = {3, 6, 9, 12, 15};
    int *p = a;
    for(int i = 0; i < 5; i++)
        cout << setw(5) << *p++;
    cout << endl;
}

```



- ```

        for(i = 0; i < 5; i++)
            cout << setw(5) << * -- p;
        cout << endl;
    }

```
6. #include <iomanip.h>
- ```

void main() {
    int a[8] = {4,8,12,16,20,24,28,32};
    int *p = a;
    do {
        cout << *p << ' ';
        p += 2;
    } while(p < a + 8);
    cout << endl;
}

```
7. #include <iomanip.h>
- ```

void main() {
    int x = 20, y = 40, *p;
    p = &x; cout << *p << ' ';
    *p = x + 10;
    p = &y; cout << *p << endl;
    *p = y + 20; cout << x << ' ' << y << endl;
}

```
8. #include <iostream.h>
- ```

void main()
{
    int x = 12, *xp = &x;
    *xp = x + 5;
    cout << *xp << ' ' << x << endl;
    x += 2 * *xp + 1;
    cout << *xp << ' ' << x << endl;
}

```
9. #include <iostream.h>
- ```

void main()
{
    char a[3][15] = {"computer", "typewriter", "telephone"};
    char(*p)[15] = a;
    for(int i = 0; i < 3; i++)
        cout << *p++ << endl; // 可把 *p++ 改写为 p[i]
}

```
10. #include <iostream.h>
- ```

#include <string.h>
typedef char AA[10];
void main()
{
    AA a, b = "camera";
    char *ap = a, *bp = b + strlen(b);
    while(bp != b) *ap++ = * -- bp;
    *ap = '\0';
}

```

```

        cout << a << ' ' << b << endl;
    }

11. #include<iostream.h>
    const n=5;
    void main()
    {
        int a[n]={3,10,5,4,7};
        int *p1=a, *p2=a+n-1;
        while(p1<p2) {
            int x=*p1; *p1=*p2; *p2=x;
            p1++; p2--;
        }
        for(int i=0;i<n;i++) cout << *(a+i) << ' ';
        cout << endl;
    }

12. #include<iostream.h>
    void main()
    {
        int x=24;
        int &y=x;
        cout << x << ' ' << y << endl;
        y=46; x=y+3;
        cout << x << ' ' << y << endl;
    }

13. #include<iostream.h>
    void main()
    {
        int x=23;
        int *p1=new int(5);
        int *p2=new int(x-8);
        cout << *p1 << ' ' << *p2 << endl;
        int temp=*p1;
        *p1=*p2;
        *p2=temp;
        cout << *p1 << ' ' << *p2 << endl;
        delete p1; delete p2;
    }

14. #include<iomanip.h>
    const int n=12;
    void main()
    {
        int *a=new int[n];
        a[0]=0; a[1]=1;
        for(int i=2; i<n; i++) a[i]=a[i-1]+a[i-2];
        for(i=0; i<n; i++) {
            cout << setw(5) << a[i];
            if((i+1)%6==0) cout << endl;
        }
        delete []a;
    }

```

• 150 •

## 第六章 函 数

一个C++语言程序由若干个程序文件和头文件所组成,每个头文件中通常带有用户类型的定义、符号常量的定义、函数的声明等内容,每个程序文件由若干个函数定义所组成,其中必有一个并且只有一个程序文件中包含有主函数 main,称此程序文件为主程序文件。函数是C++程序中的基本功能模块和执行单元,这一章专门讨论函数的定义和调用,变量的作用域和生存期等内容。

### 6.1 函数的定义

#### 6.1.1 定义格式

<类型名> <函数名> (<参数表>) <函数体>

<类型名>为系统或用户已定义的一种数据类型,它是函数执行过程中通过 return 语句要求返回的值的类型,又称为该函数的类型。当一个函数不需要通过 return 语句返回一个值时,称为无返回值函数或无类型函数,此时需要使用保留字 void 作为类型名。当类型名为 int 时,可以省略不写,但为了清楚起见,还是写明为好。

<函数名>是用户为函数所起的名字,它是一个标识符,应符合C++标识符的一般命名规则,用户通过使用这个函数名和实参表可以调用该函数。

<参数表>又称形式参数表,它包含有任意多个(含0个,即没有)参数说明项,当多于一个时其前后两个参数说明项之间必须用逗号分开。每个参数说明项由一种已定义的数据类型和一个变量标识符组成,该变量标识符成为该函数的形式参数,简称形参,形参前面给出的数据类型称为该形参的类型。一个函数定义中的<参数表>可以被省略,表明该函数为无参函数,若<参数表>用 void 取代,则也表明是无参函数,若<参数表>不为空,同时又不是保留字 void,则称为带参函数。

<函数体>是一条复合语句,它以左花括号开始,到右花括号结束,中间为一条或若干条C++语句。

在一个函数的参数表中,每个参数可以为任一种数据类型,包括普通类型、指针类型、数组类型、引用类型等,一个函数的返回值可以是除数组类型之外的任何类型,包括普通类型、指针类型和引用类型等。另外,当不需要返回值时,应把函数定义为 void 类型。

#### 6.1.2 定义格式举例

(1) void f1() {...}

```

(2) void f2(int x) {...}
(3) int f3(int x, int * p) {...}
(4) char * f4(char a[]) {...}
(5) int f5(int& x, double d) {...}
(6) int& f6(int b[10], int n) {...}
(7) void f7(float c[][N], int m, float& max) {...}
(8) bool f8(ElemType * &bt, ElemType& item) {...}

```

在第一条函数定义中,函数名为 f1,函数类型为 void,参数表为空,此函数是一个无参无类型函数。若在 f1 后面的圆括号内写入保留字 void,也表示为无参函数。

在第二条函数定义中,仅带有一个类型为 int 的形参变量 x,该函数没有返回值。

在第三条函数定义中,函数名为 f3,函数类型为 int,函数参数为 x 和 p,其中 x 为 int 型普通参数,p 为 int \* 型指针参数。

在第四条函数定义中,函数名为 f4,函数类型为 char \*,即字符指针类型,参数表中包含一个一维字符数组参数。

**注意:**在定义任何类型的一维数组参数时,不需要给出维的尺寸,当然给出也是允许的,但没有任何意义。

在第五条函数定义中,函数名为 f5,返回类型为 int,该函数带有两个形参,一个为整型引用变量 x,另一个为双精度变量 d。

在第六条函数定义中,函数名为 f6,函数类型为 int&,即整型引用,该函数带有两个形参,一个是整型数组 b,另一个是整型变量 n。在这里定义形参数组 b 所给出的数组长度 10 不起任何作用。

在第七条函数定义中,函数名为 f7,无函数类型,参数表中包含三个参数,一个为二维单精度型数组 c,第二个为整型变量 m,第三个为单精度引用变量 max。

**注意:**当定义一个二维数组参数时,第二维的尺寸必须给出,并且必须是一个常量表达式,第一维尺寸可给出也可不给出,其作用相同。

在第八条函数定义中,函数名为 f8,返回类型为 bool,即逻辑类型,该函数带有两个参数,一个为形参 bt,它为 ElemType 的指针引用类型,另一个为形参 item,它是 ElemType 的引用类型,其中 ElemType 为一种用户定义的类型或是通过 typedef 语句定义的一个类型的别名。

### 6.1.3 有关函数定义的几点说明

#### 1. 函数原型语句

在一个函数定义中,函数体之前的所有部分称为函数头,它给出了该函数的返回类型、每个参数的次序和类型等函数原型信息,所以当没有专门给出函数原型说明语句时,系统就从函数头中获取函数原型信息。

一个函数必须先定义或声明而后才能被调用,否则编译程序无法判断该函数调用的正

确性。一个函数的声明是通过使用一条函数原型语句实现的,当然使用多条相同的原型语句声明同一个函数虽然多余但也是允许的,编译时不会出现错误。

在一个完整的程序中,函数的定义和函数的调用可以在同一个程序文件中,也可以处在不同的程序文件中,但必须确保函数原型语句与函数调用表达式出现在同一个文件中,并且函数原型语句出现在前,函数的调用出现在后。

通常把一个程序中用户定义的所有函数的原型语句组织在一起,构成一个头文件,让该程序中所含的每个程序文件的开始(即所有函数定义之前)包含这个头文件(通过 # include 命令实现),这样不管每个函数的定义在哪里出现,都能够确保函数先声明后使用(即调用)这一原则的实现。

一个函数的原型语句就是其函数头的一个拷贝,当然要在最后加上语句结束符分号。函数原型语句与函数头也有细微的差别,在函数原型语句中,其参数表中的每个参数允许只保留参数类型,而省略参数名,并且若使用参数名也允许与函数头中对应的参数名不同。

## 2. 常量形参

在定义一个函数时,若只允许函数体访问一个形参的值,不允许修改它的值,则应把该形参说明为常量,这只要在形参说明的前面加上 const 保留字进行修饰即可。如:

```
void f9(const int& x, const char& y);
void f10(const char* p, char key);
```

在函数 f9 的函数体中只允许使用 x 和 y 的值,不允许修改它们的值。在函数 f10 的函数体中只允许使用 p 所指向的字符对象或字符数组对象的值,不允许修改它们的值,但在函数体中既允许使用也允许修改形参 key 的值。

## 3. 默认参数

在一个函数定义中,可根据需要对参数表末尾的一个或连续若干个参数给出默认值,当调用这个函数时,若实参表中没有给出对应的实参,则形参将采用这个默认值。如:

```
void f11(int x, int y=0) {...}
int f12(int a[], char op='+', int k=10) {...}
```

函数 f11 的定义带有两个参数,分别为整型变量 x 和 y,并且 y 带有默认值 0,若调用该函数的表达式为 f11(a,b),将把 a 的值赋给 x,把 b 的值赋给 y,接着执行函数体;若调用该函数的表达式为 f11(a+b),则也是正确的调用格式,它将把 a+b 的值赋给 x,因 y 没有对应的实参,将采用默认值 0,参数传送后接着执行函数体。

函数 f12 的定义带有三个参数,其中后两个带有默认值,所以调用它的函数格式有三种,一种只带一个实参,用于向形参 a 传送数据,后两个形参采用默认值,第二种带有两个实参,用于分别向形参 a 和 op 传送数据,第三个形参采用默认值,第三种带有三个实参,分别用于传递给三个形参。

若一个函数带有专门的函数原型语句,则形参的默认值只能在该函数原型语句中给出,不允许在函数头中给出。如对于上述的 f11 和 f12 函数,其对应的函数原型语句分别为:

```
void f11(int x, int y=0);
```

```
int f12(int a[], char op = '+', int k = 10);
```

函数定义应分别改写为:

```
void f11(int x, int y) {...}  
int f12(int a[], char op, int k) {...}
```

#### 4. 数组参数

在函数定义中的每个数组参数实际上是指向元素类型的指针参数。对于一维数组参数说明:

<数据类型> <数组名> []

它与下面的指针参数说明完全等价:

<数据类型> \* <指针变量名>

其中<指针变量名>就是数组参数说明中的<数组名>。如对于 f12 函数定义中的数组参数说明 int a[], 等价于指针参数说明 int \* a。也就是说,数组参数说明中的数组名 a 是一个类型为 int \* 的形参。

注意:在变量定义语句中定义的数组,其数组名代表的是一个数组,它的值是指向第一个元素的指针常量,这与数组形参的含义有区别。

对于二维数组参数说明:

<数据类型> <参数名> [] [<第二维尺寸>]

它与下面的指针参数说明完全等价:

<数据类型> (\* <参数名>) [<第二维尺寸>]

如对于 f7 函数定义中的二维数组参数说明 float c[][N], 等价于指针参数说明 float (\* c)[N]。

#### 5. 函数类型

当调用一个函数时就执行一遍循环体,对于类型为非 void 的函数,函数体中至少必须带有一条 return 语句,并且每条 return 语句必须带有一个表达式,当执行到任一条 return 语句时,将计算出它的表达式的值,结束整个函数的调用过程,把这个值作为所求的函数值带回到调用位置,参与相应的运算;对于类型为 void 的函数,它不需要返回任何函数值,所以在函数体中既可以使用 return 语句,也可以不使用,对于使用的每条 return 语句不允许也不需要带有表达式,当执行到任一条 return 语句时,或执行到函数体最后结束位置时,将结束函数的调用过程,返回到调用位置向下继续执行。

#### 6. 内联函数

当在一个函数的定义或声明前加上关键字 inline 则就把该函数声明为内联函数。计算机在执行一般函数的调用时,无论该函数多么简单或复杂,都要经过参数传递、执行函数体

和返回等操作。若把一个函数声明为内联函数后,在程序编译阶段系统就有可能把所有调用该函数的地方都直接替换为该函数的执行代码,由此省去函数调用时的参数传递和返回操作,从而加快整个程序的执行速度。通常可把一些相对简单的函数声明为内联函数,对于较复杂的函数则不应声明为内联函数。从用户的角度看,调用内联函数和一般函数没有任何区别。下面就是一个内联函数定义的例子,它返回形参值的立方。

```
inline int cube(int n)
{
    return n*n*n;
}
```

## 6.2 函数的调用

### 6.2.1 调用格式

调用一个已定义或声明的函数需要给出相应的函数调用表达式,其格式为:

<函数名> (<实参表>)

若调用的是一个无参函数,或全部形参为可选的函数,则<实参表>被省略,此时实参表为空。

<实参表>为一个或若干个用逗号分开的表达式,表达式的个数应至少等于不带默认值的形参的个数,应不大于所有形参的个数,<实参表>中每个表达式称为一个实参,每个实参的类型必须与相应的形参类型相同或兼容(即能够被自动转换为形参的类型,如整型与字符型就是兼容类型)。每个实参是一个表达式,包括是一个常量、一个变量、一个函数调用表达式,或一个带运算符的一般表达式。如:

- (1) g1(25)                   // 实参是一个整数
- (2) g2(x)                   // 实参是一个变量
- (3) g3(a, 2 \* b + 3)       // 第一个为变量,第二个运算表达式
- (4) g4(sin(x), '@')       // 第一个为函数调用表达式,第二个为字符常量
- (5) g5(&d, \*p, x/y - 1)   // 分别为取地址运算、间接访问和一般运算表达式

任一个函数调用表达式都可以单独作为一条表达式语句使用,但当该函数调用带有返回值时,这个值被自动丢失。对于具有返回值的函数,调用它的函数表达式通常是作为一个数据项使用,用返回值参与相应的运算,如把它赋值给一个变量,把它输出到屏幕上显示出来等。如:

- (1) f1();                   // 作为单独的语句,若有返回值则被丢失
- (2) y = f3(x, a);           // 返回值被赋给 y 保存
- (3) cout << f6(c, 10) << endl;   // 返回值被输出到屏幕上
- (4) f2(f5(x1, d1) + 1);   // f2 调用做为单独的语句,  
                                  // f5 调用是 f2 实参表达式中的一个数据项
- (5) f6(b, 5) = 3 \* w - 2;   // f6 函数调用的返回值当作一个左值

```
(6) if(f8(ct,x)) cout << "true" << endl; // f6 函数调用做为一个判断条件,
// 若返回值不为 0 则执行后面的输出语句,否则不执行任何操作
```

### 6.2.2 调用过程

当调用一个函数时,整个调用过程分为三步进行,第一步是参数传递,第二步是函数体执行,第三步是返回,即返回到函数调用表达式的位置。

参数传递称为实虚结合,即实参向形参传递信息,使形参具有确切的含义(即具有对应的存储空间和初值)。这种传递又分为两种不同情况,一种是向非引用参数传递,另一种是向引用参数传递。

形参表中的非引用参数包括普通类型的参数、指针类型的参数和数组类型的参数三种。实际上可以把数组类型的参数归为指针类型的参数。

当形参为非引用参数时,实虚结合的过程为:首先计算出实参表达式的值,接着给对应的形参变量分配一个存储空间,该空间的大小等于该形参类型的长度,然后把已求出的实参表达式的值存入到为形参变量分配的存储空间中,成为形参变量的初值。这种传递是把实参表达式的值传送给对应的形参变量,称这种传递方式为“按值传递”。

假定有下面的函数原型:

- (1) void h1(int x, int y);
- (2) bool h2(char \* p);
- (3) void h3(int a[], int n);
- (4) char \* h4(char b[][N], int m);

若采用如下的函数调用:

- (1) h1(a,25); // 假定 a 为 int 型
- (2) bool bb = h2(sp); // 假定 sp 为 char \* 型
- (3) h3(b,10); // 假定 b 为 int \* 型
- (4) char \* s = h4(c,n+1); // 假定 c 为 int(\*)[N]型,n 为 int 型

当执行第一条语句中的 h1(a,25)调用时,把第一个实参 a 的值传送给对应形参 x 的存储空间,成为 x 的初值,把常数 25 传送给形参 y 的存储空间,成为 y 的初值。

当执行第二条语句中的 h2(sp)调用时,将把 sp 的值,即一个字符对象的存储地址传送给对应的指针形参 p 的存储空间中,使 p 指向的对象就是实参 sp 所指向的对象,即 \*p 和 \*sp 指的是同一个对象,若在函数体中对 \*p 进行了修改,则待调用结束返回后通过访问 \*sp 就得到了这个修改。

当执行第三条语句中的 h3(b,10)调用时,将把 b 的值(通常为元素类型为 int 的一维数组的首地址)传送给对应数组变量(实际为指针变量)a 的存储空间中,使得形参 a 指向实参 b 所指向的数组空间,因此,在函数体中对数组 a 的存取元素的操作就是对实参数组 b 的操作。也就是说,采用数组传送能够在函数体中使用形参数组访问对应的实参数组。

当执行第四条语句中的 h4(c,n+1)调用时,将把 c 的值(通常为与形参 b 具有相同元素类型和列数的二维数组的首地址)传送给对应二维数组参数(实际为指针变量)b 的存储空间中,使得形参 b 指向实参 c 所指向的二维数组空间,在函数体中对数组 b 的存取元素的操



作就是对实参数组  $c$  的操作;该函数调用还要把第二个实参表达式  $n+1$  的值传送给形参  $m$  中,在函数体中对  $m$  的操作与相应的实参无关。

在函数定义的形参表中说明一个数组参数时,通常还需要说明一个整型参数,用它来接收由实参传送来的数组的长度,这样才能够使函数知道待处理数组中元素的个数。

当形参为引用参数时,对应的实参通常是一个变量,实虚结合的过程为:把实参变量的地址传送给引用形参,成为引用形参的地址,也就是说使得引用形参是实参变量的一个引用(别名),引用形参所占用的存储空间就是实参变量所占用的存储空间。因此,在函数体中对引用形参的操作实际上就是对被引用的实参变量的操作。这种向引用参数传递信息的方式称为引用传送或按址传送。

引用传送的好处是不需要为形参分配新的存储空间,从而节省存储,另外能够使对形参的操作反映到实参上,函数被调用结束返回后,能够从实参中得到函数对它的处理结果。有时,既为了使形参共享实参的存储空间,又不希望通过形参改变实参的值,则应当把该形参说明为常量引用,如:

```
void f13(const int& A, const Node* & B, char C);
```

在该函数执行时,只能读取引用形参  $A$  和  $B$  的值,不能够修改它们的值。因为它们是对应实参的别名,所以,只允许该函数使用  $A$  和  $B$  对应实参的值,不允许进行修改,从而杜绝了对实参进行的有意或无意的破坏。

进行函数调用除了要把实参传递给形参外,系统还将自动把函数调用表达式执行后的位置(称为返回地址)传递给被调用的函数,使之保存起来,当函数执行结束后,将按照所保存的返回地址返回到原来位置,继续向下执行。

函数调用的第二步是执行函数体,实际上就是执行函数头后面的一条复合语句,它将按照从上向下、从左向右的次序执行函数体中的每条语句,当碰到 `return` 语句时就结束返回。对于无类型函数,当执行到函数体最后的右花括号时,与执行一条不带表达式的 `return` 语句相同,也将结束返回。

函数调用的第三步是返回,这实际上是执行一条 `return` 语句的过程。当 `return` 语句不带有表达式时,其执行过程为:按函数中所保存的返回地址返回到调用函数表达式的位置接着向下执行。当 `return` 语句带有表达式时,又分为两种情况,一种是函数类型为非引用类型,则计算出 `return` 表达式的值,并把它保存起来,以便返回后访问它参与相应的运算;另一种情况是函数的类型为引用类型,则 `return` 中的表达式必须是一个左值,并且不能是本函数中的局部变量(关于局部变量的概念留在下一节讨论),执行 `return` 语句时就返回这个左值,也可以说函数的返回值是该左值的一个引用。因此,返回为引用的函数调用表达式既可作为右值又可作为左值使用,但非引用类型的函数表达式只能作为右值使用。例如:

```
int& f14(int a[], int n)
{
    int k=0;
    for(int i=1; i<n; i++)
        if(a[i]>a[k]) k=i;
    return a[k];
}
```

该函数的功能是从一维整型数组  $a[n]$  中求出具有最大值的元素并引用返回。当调用该函数时,其函数表达式既可以作为右值,从而取出  $a[k]$  的值,又可以作为左值,从而向  $a[k]$  赋予新值。如:

```
#include <iostream.h>
int& f14(int a[], int n)
{
    int k=0;
    for(int i=1;i<n;i++)
        if(a[i]>a[k]) k=i;
    return a[k];
}
void main()
{
    int b[8]={25,37,18,69,54,73,62,31};
    cout << f14(b,8) << endl;
    f14(b,5)=86;
    for(int i=0;i<8;i++) cout << b[i] << ' ';
    cout << endl;
}
```

该程序的运行结果如下,请读者自行分析。

```
73
25 37 18 86 54 73 62 31
```

通常把函数定义为引用的情况较少出现,而定义为非引用(即普通类型和指针类型)的情况则常见。

### 6.2.3 函数调用举例

程序 6-1:

```
#include <iostream.h>
int xk1(int n);
void main()
{
    cout << "输入一个正整数:";
    int m;
    cin >> m;
    int sum=xk1(m)+xk1(2*m+1);
    cout << sum << endl;
}
int xk1(int n)
{
    int i,s=0;
    for(i=1;i<=n;i++) s+=i;
    return s;
}
```

该程序包含一个主函数和一个 xk1 函数,在程序开始给出了一条 xk1 函数的原型语句,使得 xk1 函数无论在什么地方定义,在此程序文件中的所有函数都能够合法地调用它。

**注意:**主函数不需要使用相应的函数原型语句加以声明,因为C++规定不允许任何函数调用它,它只由操作系统调用并返回操作系统。

函数 xk1 的功能是求出自然数 1 至 n 之和,这个和就是 s 的最后值,由 return 语句把它返回。在主函数中首先为 m 输入一个自然数,接着用 m 去调用 xk1 函数返回 1 至 m 之间的所有自然数之和,再用  $2 * m + 1$  去调用 xk1 函数返回 1 至  $2 * m + 1$  之间的所有自然数之和,把这两个和加起来赋给变量 sum,最后输出 sum 的值。

假定从键盘上为 m 输入的正整数为 5,则进行 xk1(m)调用时把 m 的值 5 传送给 n,接着执行函数体后返回 s 的值为 15,进行 xk1( $2 * m + 1$ )调用时把  $2 * m + 1$  的值 11 传送给 n,接着执行函数体后返回 s 的值为 66,它们的和 81 被作为初值赋给 sum,最后输出的 sum 值为 81。

程序 6-2:

```
#include <iostream.h>
void xk2(int& a, int b);
void main()
{
    int x=12,y=18;
    cout <<"x=" <<x <<' ' <<"y=" <<y << endl;
    xk2(x,y);
    cout <<"x=" <<x <<' ' <<"y=" <<y << endl;
}
void xk2(int& a, int b)
{
    cout <<"a=" <<a <<' ' <<"b=" <<b << endl;
    a=a+b;
    b=a+b;
    cout <<"a=" <<a <<' ' <<"b=" <<b << endl;
}
```

该程序包含一个主函数和一个 xk2 函数,xk2 函数使用了两个形参,一个是整型引用变量 a,另一个是整型变量 b。在主函数中使用 xk2(x,y)调用时,将使形参 a 成为实参 x 的别名,在函数体中对 a 的访问就是对主函数中 x 的访问,此调用同时把 y 的值传送给形参 b,在函数体中对形参 b 的操作是与对应的实参 y 无关的,因为它们使用各自的存储空间。该程序的运行结果为:

```
x=12 y=18
a=12 b=18
a=30 b=48
x=30 y=18
```

程序 6-3:

```
#include <iostream.h>
void xk3(int * a, int * b);
```

```

void xk4(int& a, int& b);
void main()
{
    int x=5,y=10;
    cout <<"x="<<x <<' ' <<"y="<<y << endl;
    xk3(&x, &y);
    cout <<"x="<<x <<' ' <<"y="<<y << endl;
    xk4(x, y);
    cout <<"x="<<x <<' ' <<"y="<<y << endl;
}
void xk3(int * a, int * b)
{
    int c = *a;
    *a = *b;
    *b = c;
}
void xk4(int& a, int& b)
{
    int c = a;
    a = b;
    b = c;
}

```

该程序中的 xk3 函数用于交换 a 和 b 分别指向的两个对象的值,主函数使用 xk3(&x, &y)调用时,分别把 x 和 y 的地址赋给形参 a 和 b,所以实际交换的是主函数中 x 和 y 的值; xk4 函数用于直接交换 a 和 b 的值,由于 a 和 b 都是引用参数,所以在主函数使用 xk4(x,y)调用时,执行 xk4 函数实际交换的是相应实参变量 x 和 y 的值。

此程序的运行结果为:

```

x=5 y=10
x=10 y=5
x=5 y=10

```

上述的 xk3 和 xk4 具有完全相同的功能,但由于在 xk3 中使用的是指针参数,传送给它的实参也必须是对象的地址,在函数体中访问指针所指向的对象必须进行间接访问运算,所以,定义和调用 xk3 不如定义和调用 xk4 直观和简便。

程序 6-4:

```

#include <iostream.h>
const int N=8;
int xk5(int a[], int n);
void main()
{
    int b[N] = {1,7,2,6,4,5,3, -2};
    int m1 = xk5(b,8);
    int m2 = xk5(&b[2],5);
    int m3 = xk5(b+3,3);
    cout <<m1 <<' ' <<m2 <<' ' <<m3 << endl;
}
int xk5(int a[], int n)

```

```

{
    int i, f = 1;
    for(i = 0; i < n; i++) f *= a[i]; // 或写成 f *= *a++;
    return f;
}

```

该函数包含一个主函数和一个 xk5 函数, xk5 函数的功能是求出一维整型数组 a[n] 中所有元素之积并返回。在主函数中第一次调用 xk5 函数时, 把数组 b 的首地址传送给 a, 把数组 b 的长度 8 传送给 n, 执行函数体对数组 a 的操作实际上就是对主函数中数组 b 的操作, 因为它们同时指向数组 b 的存储空间; 第二次调用 xk5 函数是把数组 b 中 b[2] 元素的地址传送给 a, 把整数 5 传送给 n, 执行函数体对数组 a[n] 的操作实际上是对数组 b 中 b[2] 至 b[6] 之间元素的操作; 第三次调用 xk5 函数是把数组 b 中 b[3] 元素的地址传送给 a, 把整数 3 传送给 n, 执行函数体对数组 a[n] 的操作实际上是对数组 b 中 b[3] 至 b[5] 之间元素的操作。该程序的运行结果为:

- 10080 720 120

#### 程序 6-5:

```

#include <iostream.h>
char* xk6(char* sp, char* dp);
void main()
{
    char a[15] = "abcaedecaxybcw";
    char b[15];
    char* c1 = xk6(a, b);
    cout << c1 << ' ' << a << ' ' << b << endl;
    char* c2 = xk6(a + 4, b);
    cout << c2 << ' ' << a << ' ' << b << endl;
}

char* xk6(char* sp, char* dp)
{
    if(*sp == '\0') { *dp = '\0'; return dp; }
    int i = 0, j;
    for(char* p = sp; *p; p++) { // 扫描 sp 所指字符串中的每个字符位置
        for(j = 0; j < i; j++)
            if(*p == dp[j]) break; // 当 *p 与 dp[0] 至 dp[i-1] 之间的
            // 任一元素相同则比较过程结束
        if(j >= i) dp[i++] = *p;
        // 若 dp 数组的前 i 个元素均不等于 *p, 则把 *p 写入 dp[i] 元素中
    }
    dp[i] = '\0'; // 写入字符串结束符
    return dp;
}

```

xk6 函数的功能是把 sp 所指向的字符串, 去掉重复字符后拷贝到 dp 所指向的字符数组中, 并返回 dp 指针。在主函数中第一次调用 xk6 函数时, 分别以 a 和 b 作为实参, 第二次调用时分别以 a + 4 (即 a[4] 的地址) 和 b 作为实参。该程序运行后的输出结果为:

abcdexyw abcaedecaxybcw abcdexyw

decaxybw abcadecaxybcw decaxybw

程序 6-6:

```
#include<iostream.h>
int * xk7(int *&a1, int * a2);
int * xk7(int *&a1, int * a2)
{
    cout <<"when enter xk7: * a1, * a2 ="<< * a1 <<" , "<< * a2 << endl;
    a1 = new int(2 * * a1 + 4);
    a2 = new int(2 * * a2 - 1);
    cout <<"when leave xk7: * a1, * a2 ="<< * a1 <<" , "<< * a2 << endl;
    return a2;
}
void main()
{
    int x=10, y=25;
    int * xp = &x, * yp = &y;
    cout <<"before call xk7: * xp, * yp ="<< * xp <<" , "<< * yp << endl;
    int * ip = xk7(xp,yp);
    cout <<"after call xk7: * xp, * yp ="<< * xp <<" , "<< * yp << endl;
    cout <<"* ip ="<< * ip << endl;
    delete xp; // xp 指向的是在执行 xk7 函数时动态分配的对象 * a1
    delete ip; // ip 指向的是在执行 xk7 函数时动态分配的对象 * a2
}
```

在 xk7 函数的定义中,把形参 a1 定义为整型指针的引用,把 a2 定义为整型指针,当在主函数中利用 xk7(xp,yp)表达式调用该函数时,a1 就成为 xp 的别名,访问 a1 就等于访问主函数中的 xp,而 a2 同 yp 具有各自独立的存储空间,a2 的初值为 yp 的值,在 xk7 函数中对 a2 的访问(指直接访问)与 yp 无关。此程序运行结果为:

```
before call xk7: * xp, * yp = 10, 25
when enter xk7: * a1, * a2 = 10, 25
when leave xk7: * a1, * a2 = 24, 49
after call xk7: * xp, * yp = 24, 25
* ip = 49
```

## 6.3 变量的作用域

在一个 C++ 程序中,对于每个变量必须遵循先定义后使用的原则。根据变量定义的位置不同将使它具有不同的作用域。一个变量离开了它的作用域,在定义时为它分配的存储空间就被系统自动回收了,因此该变量也就不存在了。

### 6.3.1 作用域分类

变量的作用域具有四种类别:全局作用域、文件作用域、函数作用域和块作用域。具有

全局作用域的变量称为全局变量,具有块作用域的变量称为局部变量。

### 1. 全局作用域

当一个变量在一个程序文件的所有函数定义之外(并且通常在所有函数定义之前)定义时,则该变量具有全局作用域,即该变量在整个程序包括的所有文件中都有效,都是可见的,都是可以访问的。当一个全局变量不是在本程序文件中定义时,若要在本程序文件中使用,则必须在本文件开始进行声明,声明格式为:

```
extern <类型名> <变量名>, <变量名>, ...;
```

它与变量定义语句格式类似,其区别是:不能对变量进行初始化,并且要在整个语句前加上 `extern` 保留字。

当用户定义一个全局变量时,若没有对其初始化,则编译时会自动把它初始化为 0。

### 2. 文件作用域

当一个变量定义语句出现在一个程序文件中的所有函数定义之外,并且该语句前带有 `static` 保留字时,则该语句定义的所有变量都具有文件作用域,即在整个程序文件中有效,但在其他文件中是无效的,不可见的。

若在定义文件作用域变量时没有初始化,则编译时会自动把它初始化为 0。

### 3. 函数作用域

在每个函数中使用的语句标号具有函数作用域,即它在本函数中有效,供本函数中的 `goto` 语句跳转使用。由于语句标号不是变量,应该说函数作用域不属于变量的一种作用域。

### 4. 块作用域

当一个变量是在一个函数体内定义时,则称它具有块作用域,其作用域范围是从定义点开始,直到该块结束(即所在复合语句的右花括号)为止。

具有块作用域的变量称为局部变量,若局部变量没有被初始化,则系统也不会对它初始化,它的初值是不确定的。对于在函数体中使用的变量定义语句,若在其前面加上 `static` 保留字,则称所定义的变量为静态局部变量,若静态局部变量没有被初始化,则编译时会被自动初始化为 0。

对于非静态局部变量,每次执行到它的定义语句时,都会为它分配对应的存储空间,并对带初值表达式的变量进行初始化;而对于静态局部变量,只是在整个程序执行过程中第一次执行到它的定义语句时为其分配对应的存储空间,并进行初始化,以后再执行到它时什么都不会做,相当于第一次执行后就删除了该语句。

任一函数定义中的每个形参也具有块作用域,这个块是作为函数体的复合语句,当离开函数体后它就不存在了,函数调用时为它分配的存储空间也就被系统自动回收了,当然引用参数对应的存储空间不会被回收。由于每个形参具有块作用域,所以它也是局部变量。

在 C++ 程序中定义的符号常量也同变量一样具有全局、文件和局部这三种作用域。当符号常量定义语句出现在所有函数定义之外,并且在前面带有 `extern` 保留字时,则所定义的

常量具有全局作用域,若在前面带有 `static` 关键字或什么都没有,则所定义的常量具有文件作用域。若符号常量定义语句出现在一个函数体内,则定义的符号常量具有局部作用域。

一个C++ 程序中的所有函数的函数名都具有全局作用域,所以在程序中所含的任何文件内都可以使用任一函数名构成函数调用表达式,执行对应的函数。

具有同一作用域的任何标识符,不管它表示什么对象(如常量、变量、函数、类型等)都不允许重名,若重名系统就无法惟一确定它的含义了。

由于每一个复合语句就是一个块,所以在不同复合语句中定义的对象具有不同的块作用域,也称为具有不同的作用域,其对象名允许重名,因为系统能够区分它们。

### 6.3.2 程序举例

程序 6-7:

程序主文件 6-7.cpp

```
#include <iostream.h>
int xk8(int n);          // 函数 xk8 的原型声明
int xk9(int n);          // 函数 xk9 的原型声明
int AA=5;                // 定义全局变量 AA
extern const int BB=8;    // 定义全局常量 BB
static int CC=12;        // 定义文件域变量 CC
const int DD=23;         // 定义文件域常量 DD
void main()
{
    int x=15;             // x 的作用域为主函数体
    cout << "x * x = " << xk8(x) << endl;
    cout << "mainFile: AA, BB = " << AA << ', ' << BB << endl;
    cout << "mainFile: CC, DD = " << CC << ', ' << DD << endl;
    cout << xk9(16) << endl;
}
int xk9(int n)            // n 的作用域为 xk9 函数体
{
    int x=10;             // x 的作用域为 xk9 函数体
    cout << "xk9: x = " << x << endl;
    return n * x;
}
```

程序次文件 6-7-1.cpp

```
#include <iostream.h>
int xk8(int n);          // 函数 xk8 的原型声明
extern int AA;           // 全局变量 AA 的声明
extern const int BB;     // 全局常量 BB 的声明
static int CC=120;       // 定义文件域变量 CC
const int DD=230;        // 定义文件域常量 DD
int xk8(int n)            // n 的作用域为 xk8 函数体
{
    cout << "attachFile: AA, BB = " << AA << ', ' << BB << endl;
    cout << "attachFile: CC, DD = " << CC << ', ' << DD << endl;
}
```



```

    return n*n;
}

```

此程序包含两个程序文件,定义有各种类型的变量和常量,其中 AA 为全局变量, BB 为全局常量, CC 为各自的文件域变量, DD 为各自的文件域常量,主函数中的 x 为作用于主函数的局部变量, xk9 函数中的 x 为作用于该函数的局部变量, xk8 和 xk9 函数的各自参数表中的形参 n 是作用于各自函数的局部变量。为了在程序次文件 6-7-1.cpp 中能够使用程序主文件 6-7.cpp 中定义的全局变量 AA 和全局常量 BB, 必须在该文件开始对他们进行声明。

当上机输入和运行该程序时,可以先建立程序主文件 6-7.cpp 并编译通过,再建立程序次文件 6-7-1.cpp 并编译通过,然后把它们连接起来生成可执行文件 6-7.exe。该程序的运行结果为:

```

attachFile: AA, BB = 5, 8
attachFile: CC, DD = 120, 230
x * x = 225
mainFile: AA, BB = 5, 8
mainFile: CC, DD = 12, 23
xk9: x = 10
160

```

请读者结合上述程序分析结果的正确性。

程序 6-8:

```

#include <iostream.h>
const int N = 10;
void main()
{
    int a[N] = {3, 8, 12, 20, 15, 6, 7, 24, 8, 19};
    for(int i = 0; i < N/2; i++) {
        int x = a[i];
        a[i] = a[N-i-1];
        a[N-i-1] = x;
    }
    for(i = 0; i < N; i++) cout << a[i] << ' ';
    cout << endl;
}

```

在这个程序中, N 为文件域常量, a 和 i 分别为主函数体复合语句块内的局部数组和变量, 它们的作用域从定义点开始到主函数结束, x 为 for 循环体复合语句块内的局部变量, 它的作用域从定义点开始到 for 循环体结束。

在主函数中, 首先定义了一维整型数组 a[N], 接着利用 for 循环交换数组 a 中前后对称元素的值, 使得 a 中的每个元素值按原有位置的逆序排列, 然后依次输出 a 中每个元素值。该程序运行结果为:

```

19 8 24 7 6 15 20 12 8 3

```

程序 6-9:

```

#include<iostream.h>
void input();
void output();
int sumSquare(int b[], int n);
const int nn=5;    // 定义文件域常量 nn
int a[nn];          // 定义全局域数组 a[nn]
void main()
{
    input();
    output();
    cout <<sumSquare(a,nn)<< endl; // 使用数组 a 和常量 nn 作为实参
}
void input()
{
    cout <<"为数组 a 输入"<<nn<<"个整数:"<< endl;
    for(int i=0;i<nn;i++) cin >>a[i]; // i 是本函数的局部变量
}
void output()
{
    cout <<"输出数组 a 中的"<<nn<<"个元素值:"<< endl;
    for(int i=0;i<nn;i++) cout <<a[i]<<' '; // i 是本函数的局部变量
    cout << endl;
}
int sumSquare(int b[], int n) // b 将指向对应的实参数组 a,
    // 形参指针 b 和形参 n 是本函数中的局部变量
{
    // 求数组 b 中 n 个元素之和的平方
    int s=0,i; // s 和 i 是本函数的局部变量
    for(i=0;i<n;i++) s += b[i];
    return s*s;
}

```

该程序包含一个主函数和三个一般函数,主函数依次调用这三个函数。Input 函数从键盘上向数组 a[nn]输入数据,output 函数依次输出数组 a[nn]中每个元素的值,sumSquare 函数求出数组 b 中 n 个元素值之和的平方。由于调用 sumSquare 函数是把实参数组 a 和常量 nn 分别传送给形参数组 b 和形参变量 n,所以在函数体中对数组 b[n]的操作实际上是对实参数组 a[nn]的操作。

在本程序中,nn 为文件域常量,a[nn]为全局域数组,所以,它们能够使用在该程序中的任何地方,即在任何地方都是可见的。

假定程序运行时从键盘上输入的 5 个整数为:1,2,3,4,5,则得到的运行结果为:

```

为数组 a 输入 5 个整数:
1 2 3 4 5
输出数组 a 中的 5 个元素值:
1 2 3 4 5
225

```

程序 6-10:

```

#include<iostream.h>
• 166 •

```

```

int x = 10;
void main()
{
    int y = 20;
    cout << "x,y=" << x << ', ' << y << endl;
    {
        int x = 30;
        y = y + x;
        cout << "x,y=" << x << ', ' << y << endl;
    }
    cout << "x,y=" << x << ', ' << y << endl;
}

```

在函数体外定义的  $x$  为全局变量,在主函数体中定义的  $y$  为作用于整个函数体的局部变量,在主函数体中的一条复合语句中又定义了一个变量  $x$ ,它的作用域只局限于该复合语句内,离开了该复合语句它就不存在了。

在C++中,当一个作用域包含另一个作用域时,则在里层作用域内可以定义与外层作用域同名的对象,此时在外层定义的同名对象,在内层将被重新定义的同名对象屏蔽掉,使之变为不可见。如在此程序主函数体中的一条复合语句内,由于重新定义了变量  $x$ ,所以全局变量  $x$  在此复合语句内暂时被屏蔽掉,当离开这条复合语句后,全局变量  $x$  为有效。此程序运行结果如下:

```

x,y = 10,20
x,y = 30,50
x,y = 10,50

```

**提示:** 若要在函数体内访问与局部变量同名的全局域或文件域变量,则只要在该变量名前加上作用域区分符( $::$ )即可。如  $::x$  使用在上述主函数中定义有  $x$  的复合语句内时,则就表示全局变量  $x$ ,若不加作用域区分符则表示在当前作用域内定义的变量  $x$ 。

程序 6-11:

```

#include <iostream.h>
int xk10(int m, int n)
    // 求出 m 和 n 的最大公约数
{
    int r = m % n;
    while(r != 0) {
        m = n;
        n = r;
        r = m % n;
    }
    return n;
}
void main()
{
    int m, n;
    do {
        cout << endl;

```

```

        cout << "输入两个整数求其最大公约数(若任一数 <= 0 则结束):";
        cin >> m >> n;
        if(m <= 0 || n <= 0) break; // 输入的任一数小于等于 0 则结束循环
        cout << m << "和" << n << "的最大公约数为:" << xk10(m,n) << endl;
    } while(1);
}

```

在这个程序中,主函数和 xk10 函数中都定义有 m 和 n 这两个整数变量,并且主函数调用 xk10 是通过值传送进行的,所以主函数中的 m 和 n 与 xk10 函数中的 m 和 n 分别占用各自的存储空间,分别具有各自的作用域,一个函数中的 m 和 n 值的变化与另一个函数中的 m 和 n 无关。假定需要依次求出(75,15),(36,90),(74,25),(350,48)等四组整数的最大公约数,则程序运行结果如下:

```

输入两个整数求其最大公约数(若任一数 <= 0 则结束):75 15
75 和 15 的最大公约数为:15

```

```

输入两个整数求其最大公约数(若任一数 <= 0 则结束):36 90
36 和 90 的最大公约数为:18

```

```

输入两个整数求其最大公约数(若任一数 <= 0 则结束):74 25
74 和 25 的最大公约数为:1

```

```

输入两个整数求其最大公约数(若任一数 <= 0 则结束):350 48
350 和 48 的最大公约数为:2

```

```

输入两个整数求其最大公约数(若任一数 <= 0 则结束):0 0

```

#### 程序 6-12:

```

#include <iostream.h>
void xk11(int& x, int y);
void main()
{
    int x = 12, y = 25;
    xk11(x,y);
    cout << "main1:x,y=" << x << ' ' << y << endl;
    xk11(y,x);
    cout << "main2:x,y=" << x << ' ' << y << endl;
    xk11(x,x+y);
    cout << "main3:x,y=" << x << ' ' << y << endl;
}
void xk11(int& x, int y)
{
    x = x + 2; y = x + y;
    cout << "xk11: x,y=" << x << ' ' << y << endl;
}

```

在 xk11 函数中,说明 x 为引用参数,y 为非引用参数,在主函数中也定义 x 和 y 变量,每次利用不同的实参调用 xk11 函数,并通过输出语句显示出 x 和 y 的值,读者可以借此分析不同的参数传递方式对不同作用域内变量的影响作用。该程序运行结果为:

```

xk11: x,y=14 39
main1:x,y=14 25
xk11: x,y=27 41
main2:x,y=14 27
xk11: x,y=16 57
main3:x,y=16 27

```

程序 6-13:

```

#include <iostream.h>
void xk12();
void main()
{
    for(int i=0;i<5;i++) xk12();
}
void xk12()
{
    int a=0; // a若不被初始化,则初值是未知的
    a++;
    static int b=0; // b若不被初始化,也将被自动赋初值 0
    b++;
    cout <<"a="<<a<<" , b="<<b<<endl;
}

```

在该程序的 xk12 函数中定义有局部变量 a 和静态局部变量 b,主函数 5 次调用这个函数,每次调用都将为 a 分配存储空间并被初始化为 0,但只有第一次调用才为 b 分配存储空间并初始化为 0,其余 4 次调用都不会再建立 b 并初始化,将始终访问第一次建立的 b,也就是说,静态局部变量同全局变量和文件域变量一样,一经建立和初始化后将在整个程序运行过程中始终存在,只有当程序运行结束时系统才收回分配给它的存储空间。该程序的运行结果为:

```

a=1, b=1
a=1, b=2
a=1, b=3
a=1, b=4
a=1, b=5

```

总之,对于全局变量、文件域变量、加入 static 保留字定义的局部变量、利用 new 运算符动态分配的对象,它们的生存期(即所占用存储空间的持续时间)从定义点开始直到整个程序执行结束(当然可随时利用 delete 运算回收动态对象);对于在任何函数体中定义的局部变量和值参(即非引用参数),它们的生存期从定义点开始到所在的复合语句块结束。

## 6.4 递归函数

在 C++ 语言程序中,主函数可以调用其他任何函数,任一函数又可以调用除主函数之外的任何函数。特别地,一个函数还可以直接或间接地调用它自己,这种情况称为直接或间接递归调用。直接递归是指在一个函数体中使用调用本函数的函数调用表达式,间接递归

是指在一个函数中调用另一个函数,而在另一个函数中又反过来调用这个函数。这里只简要讨论一下直接递归调用的情况。

若一个问题的求解可以化为较小问题的求解,而较小问题的求解又可化为更小问题的求解,依次类推,这种有规律地将原问题逐渐细化的过程,若其求解小问题的方法与求解大问题的方法相同,则称为递归求解过程。由于在递归过程中,求解的问题越化越小,最后必然能够得到一个最小问题的解,它不需要再向下递归求解,得到了这个最小问题的解后,再逐层向上返回,依次得到较大问题的解,最终必将得到原有问题的解。

例 1: 利用递归方法求解一维数组  $a[n]$  中  $n$  个元素之和。

分析: 把求解数组  $a$  中  $n$  个元素之和看做为求解数组  $a$  中  $n-1$  个元素之和,把这个和与元素  $a[n-1]$  相加就得到了原问题的解,再把求解数组  $a$  中  $n-1$  个元素之和看做为求解数组  $a$  中  $n-2$  个元素之和,把这个和与元素  $a[n-2]$  相加就得到了上一层问题的解,依次类推,直到求解数组  $a$  中 1 个元素之和时可直接得到  $a[0]$ ,从此结束逐层向下递归的过程,接着逐层向上返回,第一次返回可由一个元素之和得到两个元素之和,第二次返回再由二个元素之和得到三个元素之和,依次类推,直到第  $n-1$  次返回后根据返回值(即数组  $a$  中前  $n-1$  个元素之和)加上元素  $a[n-1]$  的值再返回就结束了整个递归求解过程。

采用递归方法编写的计算函数称为递归函数。假定  $a[n]$  数组的元素类型为 `int`,则求解数组  $a$  中  $n$  个元素之和的递归函数为:

```
int fun1(int a[], int n)
{
    if(n <= 0) {
        cerr << "参数 n 值非法!" << endl;
        exit(1);
    }
    if(n == 1) return a[0];
    else return a[n-1] + fun1(a, n-1);
}
```

在这个函数中, `fun1(a, n-1)` 为一个函数递归调用表达式,进行递归调用和普通调用(即非递归调用)一样,也经过参数传递、函数体执行和结束返回这三个步骤。在这个函数中,共需要进行  $n-1$  次递归调用,每次都要把实参  $a$  的值赋给本次递归调用为形参  $a$  分配的存储空间中,把实参  $n-1$  的值赋给本次递归调用为形参  $n$  分配的存储空间中,接着执行函数体,若当前  $n$  的值等于 1,则结束本次的递归调用,直接返回  $a[0]$  的值,并使程序执行返回到进行本次递归调用的 `return` 语句中,接着计算出  $a[n-1]$  与返回值之和,然后又向上层的调用返回,依次类推;若当前  $n$  的值大于 1,则执行 `else` 后面的 `return` 语句,接着再向下进行递归调用。

下面是计算一维数组  $b[n]$  中  $n$  个元素之和的完整程序。

```
#include <iostream.h>
#include <stdlib.h>
int fun1(int a[], int n);
void main()
{
    int b[8] = {5, 16, 7, 9, 20, 13, 18, 6};
```

```

        int s = fun1(b,8);
        cout << s << endl;
    }
    int fun1(int a[], int n)
    {
        if(n <= 0) {
            cerr << "参数 n 值非法!" << endl;
            exit(1);
        }
        if(n == 1) return a[0];
        else return a[n-1] + fun1(a,n-1);
    }
}

```

主函数中利用 fun1(b,8)调用递归函数 fun1 称为第 0 次递归调用,进行此次调用时把数组 b 的首地址传送给数组参数(又称指针参数)a,把常量 8 传送给形参 n,以便计算出数组 b 中前 8 个元素之和。当函数 fun1 被主函数调用的过程中,n 的值将在各层递归调用时从 8 依次变化到 1,if 后面的 return 语句只在最后一次递归调用时被执行并返回 a[0]的值,其余每次递归调用都执行 else 后面的 return 语句,依次返回前 2 个、3 个、...、8 个元素的值。该函数的运行结果,即 s 的值为 94。

例 2. 利用递归方法求解 n 阶乘(n!)的值。

分析:设用函数 f(n)表示 n!,由数学知识可知,n 阶乘的递归定义为:它等于 n 乘以 n-1 的阶乘,当 n 等于 0 或 1 时,函数值为 1,用数学公式表示为:

$$f(n) = \begin{cases} 1 & (n == 0 \text{ 或 } 1) \\ n * f(n-1) & (n > 1) \end{cases}$$

在这里 n 等于 0 或 1 是递归终止的条件,得到的函数值为 1,当 n 大于 1 时需要向下递归先求出 f(n-1)的值后,再乘以 n 才能够得到 f(n)的值。计算 f(n)的递归函数为:

```

int f(int n)
{
    if(n == 0 || n == 1) return 1;
    else return n * f(n-1);
}

```

假定用 f(5)去调用 f(n)函数,该函数返回 5 \* f(4)的值,因返回表达式中包含有函数 f(4)表达式,所以接着进行递归调用,返回 4 \* f(3)的值,依次类推,当最后进行 f(1)递归调用,返回函数值 1 后,结束本次递归调用,返回到调用函数 f(1)的位置,从而计算出 2 \* f(1)的值 2,即 2 \* f(1) = 2 \* 1 = 2,作为调用函数 f(2)的返回值,返回到 3 \* f(2)表达式中,计算出值 6 作为 f(3)函数的返回值,接着返回到 4 \* f(3)表达式中,计算出值 24 作为 f(4)函数的返回值,再接着返回到 5 \* f(4)表达式中,计算出 f(5)的返回值 120,从而结束整个调用过程,返回到调用函数 f(5)的位置继续向下执行。

上述调用和返回过程可形象地用图 6-1 表示。

利用上述计算 n 阶乘的函数,可以编写出一个完整程序计算出组合数  $C_m^k$ ,其中 m 和 k 均为正整数,并且  $m \geq k$ 。由数学知识可知,组合数  $C_m^k$  的含义是从 m 个互不相同的元素中每次取出 k 个不同元素所有不同取法的种数。 $C_m^k$  也可写成  $C(m,k)$ , $C_m^k$  的计算公式为:

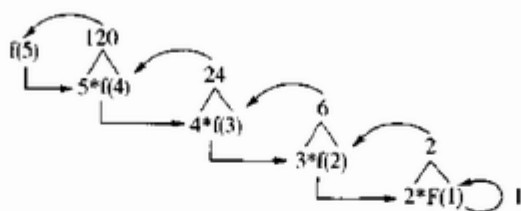


图 6-1 利用  $f(5)$  调用  $f(n)$  递归函数的执行流程

$$C_m^k = \frac{m!}{k! (m-k)!}$$

下面给出此题的完整程序,其中  $m$  和  $k$  的值由键盘输入。

```

#include <iostream.h>
int f(int n);
void main()
{
    int m, k;
    cout << "求从 m 个互不相同的元素中每次取出 k 个元素的组合数 ." << endl;
    do {
        cout << "输入 m 和 k 的值:";
        cin >> m >> k;
        if (m > 0 && k > 0 && m >= k) break;
        else cout << "输入数据不正确,重输!" << endl;
    } while (1);
    cout << "c(" << m << ", ' << k << ") = ";
    cout << float(f(m)) / (f(k) * f(m - k)) << endl;
}

int f(int n)
{
    if (n == 0 || n == 1) return 1;
    else return n * f(n - 1);
}

```

假定要求出  $C(10,3)$  的值,则程序运行结果如下:

求从  $m$  个互不相同的元素中每次取出  $k$  个元素的组合数。

输入  $m$  和  $k$  的值:10 3

$c(10,3) = 120$

对于像上述那样的递归函数都可以很方便地改写为非递归函数,求  $n$  阶乘的非递归函数如下:

```

int f(int n)
{
    int s = 1;
    for (int i = 1; i <= n; i++) s *= i;
    return s;
}

```

求数组  $a[n]$  中  $n$  个元素之和的非递归函数为:



```

int fun1(int a[], int n)
{
    if(n <= 0) {
        cerr << "参数 n 值非法!" << endl;
        exit(1);
    }
    int s = 0;
    for(int i = 0; i < n; i++) s += a[i];
    return s;
}

```

递归求解是一种非常重要的求解问题的方法,在计算机领域有着广泛的用途。当然为了说明问题,上面列举的只是最简单的例子,它们还不如非递归函数来得简单和易读。

## 6.5 函数重载

C++ 程序中的每个函数都是并列定义的,不允许在一个函数中定义另一个函数,即只允许函数嵌套调用,不允许嵌套定义。每个函数的函数名都是全局量,按理说不应该重名,若重名就是重复定义错误。但有一种情况例外,即当且仅当两个函数的参数表不同时,允许这两个函数重名(即具有相同的名字),进行函数调用时,系统会根据函数名和参数表惟一确定调用哪一个函数。当两个参数表中的任一参数的类型对应不同,或者两参数表中的参数个数不同(带有默认值的参数不算在内),则认为这两个参数表不同。

这种具有相同函数名但具有不同参数表的函数称为重载函数,允许使用相同函数名定义多个函数的情况称为函数重载。如:

- (1) void f1(int x, char h, float d = 1) { ... }
- (2) char f1() { ... } // 与(1)参数个数不同
- (3) void f1(int x) { ... } // 与(1)参数个数不同
- (4) void f1(char ch) { ... } // 与(1)参数类型和个数均不同
- (5) void f1(char ch, int x) { ... } // 与(1)参数类型不同
- (6) void f1(int a, char b, double c) { ... } // 与(1)参数类型不同
- (7) int f1(int a, int b) { ... } // 与(1)参数类型不同
- (8) double \* f1(double a[], int n) { ... } // 与(1)参数类型不同
- (9) void f1(int a, char b) { ... }
- (10) void f1(int a, char b, float c) { ... }
- (11) int f1(int x, char y) { ... }

在这些函数中,前八个函数为重载函数,因为它们的函数名相同,并且要么它们的参数个数不同,要么它们对应参数的类型不同,要么这两者均不同;第九个函数不是第一个函数的重载函数,因为对应的参数类型和个数均相同(不考虑带有默认值的参数),当在函数调用表达式中省略最后一个实参时,系统就无法惟一确定调用哪一个函数;第 10 个函数也不是第一个函数的重载函数,因为对应的参数类型和个数均相同,当在函数调用表达式中不省略最后一个实参时,系统也无法惟一确定调用哪一个函数;第 11 个函数也不是第一个函数的

重载函数,因为它只是返回类型不同,参数表中对应的参数类型和个数均相同,返回类型不同不是判断是否能够重载的条件。当然参数名不同更不是判断重载的条件。

下面程序就使用了两个重载函数,函数名为 FindMax,一个函数的功能是求出一维整型数组 a 中 n 个元素的最大值,另一个函数的功能是求出二维字符数组 a 中保存的 n 个字符串的最大值,因为这两个函数的功能相同,只是参数的类型和实现上有所不同,所以应定义为重载函数。

```
#include <iostream.h>
#include <string.h>
const int M=15;
int FindMax(int a[], int n);
char * FindMax(char a[][M], int n);
void main()
{
    int a[6]={45,28,59,43,72,36};
    char b[6][M]={"qiushuhua","wangchunfong","ningchen",
                 "zhaoyuanlin","guliang","shenyafen"};
    int x=FindMax(a,6);
    char * cp=FindMax(b,6);
    cout <<x <<' ' << cp << endl;
}
int FindMax(int a[], int n)
{
    int x=a[0];
    for(int i=1; i<n; i++)
        if(a[i]>x) x=a[i];
    return x;
}
char * FindMax(char a[][M], int n)
{
    char * x=a[0];
    for(int i=1; i<n; i++)
        if(strcmp(a[i],x)>0) x=a[i];
    return x;
}
```

该程序的运行结果为:

72 zhaoyuanlin

## 6.6 函数模板

对于普通函数,所使用的每个对象的类型都是确定的,如:

```
int max(int x, int y)
{
    return (x>y? x:y);
}
```

该函数中每个参数的类型和函数返回类型均为整型。该函数的功能是返回两个整型参数  $x$  和  $y$  中的最大值。若要求两个双精度数中的最大值则需要定义出如下函数：

```
double max(double x, double y)
{
    return (x>y? x:y);
}
```

它是上述函数的一个重载,当函数调用表达式中的两个参数均为 `int` 型时则自动调用第一个重载函数,当这两个参数均为 `double` 型时则自动调用第二个重载函数。

若能够把功能相同只是类型不同的多个重载函数用一个函数来描述,将会给程序设计带来极大的方便。在 C++ 中可以通过定义函数模板来实现。每个函数模板中可以定义一个或若干个类型参数,每个类型参数代表一种数据类型,该数据类型由进行函数调用时决定,函数模板中可以利用这些类型参数定义函数返回类型,参数类型和函数体中的变量类型。

函数模板的定义格式为：

```
template< <类型参数表> > <返回类型> <函数名> (<函数形参表>) {...}
```

<类型参数表> 中包含一个或多个用逗号分开的类型参数项,每一项由保留字 `class` 开始,后跟一个用户命名的标识符,此标识符为类型参数,表示一种数据类型,它可以同一般数据类型一样使用在函数中的任何地方。

<函数形参表> 必须至少给出一个参数说明,并且在 <类型参数表> 中给出的每个类型参数都必须在 <函数形参表> 中得到使用,即作为形参的类型使用。

下面给出一些函数模板定义的例子。

格式举例 1:

```
template<class T> T max(T x, T y)
{
    return (x>y?x:y);
}
```

此函数模板定义了  $T$  为一种类型参数,用  $T$  作为函数的返回类型以及  $x$  和  $y$  参数的类型。该函数模板的功能是返回类型为  $T$  的  $x$  和  $y$  中的最大值。模板中  $T$  的具体类型由调用它的函数表达式决定。

格式举例 2:

```
template<class A, class B> void ff(A a, B b)
{
    cout << a << ' ' << b << endl;
    cout << sizeof(a) << ' ' << sizeof(b) << endl;
}
```

此函数模板定义了  $A$  和  $B$  两个类型参数,用  $A$  作为形参  $a$  的类型,用  $B$  作为形参  $b$  的类型。该函数模板的功能是显示出  $a$  和  $b$  的值及相应的类型长度。同样,  $A$  和  $B$  的具体类型由调用它的函数表达式决定。

格式举例 3:

```

template< class Type> void inverse(Type a[], int n)
{
    Type x; int i;
    for(i = 0; i < n/2; i++) {
        x = a[i];
        a[i] = a[n - i - 1];
        a[n - i - 1] = x;
    }
}

```

该函数模板定义了 Type 为一种类型参数,用该类型定义形参数组 a 和函数体中的变量 x。该模板的功能是使数组 a 中的 n 个元素的值按逆序排列。

函数模板的原型语句也是由它的函数头后加分号所组成。如上述三个函数模板的原型语句分别如下:

- (1) template< class T> T max(T x, T y);
- (2) template< class A, class B> void f(A a, B b);
- (3) template< class Type> void inverse(Type a[], int n);

调用函数模板的表达式同调用一般函数的表达式的格式相同,由函数名和实参表所组成,如可以使用 max(a,b) 调用函数模板 max,当 a 和 b 均为 int 型时,则自动为类型参数 T 赋予 int 类型,当 a 和 b 均为 double 型时,则自动为类型参数 T 赋予 double 类型。总之,函数模板中的每个类型参数将在调用时赋予具有该类型的形参所对应的实参的类型。

当利用一个函数调用表达式调用一个函数模板时,系统首先确定类型参数所对应的具体类型,并按该类型生成一个具体函数,然后再调用这个具有确定类型的具体函数。由函数模板在调用时生成的具体函数,称为模板函数,它是函数模板的一个实例。如利用 max(a, b)调用函数模板 max 时,假定 a 和 b 均为 int 型实参,则由系统自动生成的模板函数为:

```

int max(int x, int y)
{
    return (x > y ? x : y);
}

```

若利用 inverse(b,10)调用对应的函数模板,并假定实参数组 b 中的元素类型为 double,则由系统自动生成的模板函数为:

```

void inverse(double a[], int n)
{
    double x; int i;
    for(i = 0; i < n/2; i++) {
        x = a[i];
        a[i] = a[n - i - 1];
        a[n - i - 1] = x;
    }
}

```

在一个程序中,当进行函数调用时若存在对应的一般函数(即非模板函数),则将优先调用这个一般函数,只有当不存在对应的一般函数时,才会由对应的函数模板生成模板函数,然后调用之。如假定在一个程序中既存在 max 函数模板的定义,又存在如下的一个重载函

数的定义:

```
char * max(char * x, char * y)
{
    return(strcmp(x,y) > 0)?x:y;
}
```

当使用 `max(a,b)` 进行函数调用时,并假定 `a` 和 `b` 均为 `char*` 类型,则系统将优先调用非模板的 `max` 函数,而不会去调用由 `max` 函数模板生成的、类型为 `char*` 的模板函数。若没有专门给出类型为字符指针的 `max` 函数,将调用由函数模板生成的函数,此时比较的只是两个指针的值,而达不到比较两个指针所指字符串的目的,这种调用将是不正确的。因此,对于函数模板中特殊类型的处理,必须再给出相应的一般函数的定义,该函数是带有具体类型的函数模板的一个重载函数。

在调用函数模板时,类型参数是根据该类型的形参所对应的实参的类型自动确定的,但也可以由用户在函数调用表达式中显式给出。即在函数名和实参表之间用一对尖括号把一种或若干个用逗号分开的实际类型括起来。如函数调用 `max<int>(a,b)` 将使 `max` 函数模板生成一个类型参数 `T` 为 `int` 的模板函数,不管 `a` 和 `b` 的类型如何,都将会把它们的值转换为整数后再传送给对应的整型参数 `x` 和 `y`。

下面给出一个使用函数模板的程序例子,供读者分析。

```
#include <iomanip.h>
#include <string.h>
template<class TT> void swop(TT& x, TT& y);
void swop(char * x, char * y);
void main()
{
    int a1=20, a2=35;
    double b1=3.25, b2=-4.86;
    char c1='a', c2='b';
    char d1[10]="abcdef", d2[10]="ghijk";
    cout.setf(ios::left); // 使输出的数据在显示区域内靠左显示
    cout << "数据交换前:" << endl;
    cout << "a1=" << setw(10) << a1 << "a2=" << setw(10) << a2 << endl;
    cout << "b1=" << setw(10) << b1 << "b2=" << setw(10) << b2 << endl;
    cout << "c1=" << setw(10) << c1 << "c2=" << setw(10) << c2 << endl;
    cout << "d1=" << setw(10) << d1 << "d2=" << setw(10) << d2 << endl;
    swop(a1,a2);
    swop(b1,b2);
    swop(c1,c2);
    swop(d1,d2);
    cout << endl << "数据交换后:" << endl;
    cout << "a1=" << setw(10) << a1 << "a2=" << setw(10) << a2 << endl;
    cout << "b1=" << setw(10) << b1 << "b2=" << setw(10) << b2 << endl;
    cout << "c1=" << setw(10) << c1 << "c2=" << setw(10) << c2 << endl;
    cout << "d1=" << setw(10) << d1 << "d2=" << setw(10) << d2 << endl;
}
template<class TT> void swop(TT& x, TT& y)
{
```

```

    TT w = x; x = y; y = w;
}
void swop(char * x, char * y)
{
    int n = strlen(x);
    char * w = new char[n + 1];
    strcpy(w, x);
    strcpy(x, y);
    strcpy(y, w);
}

```

该程序的运行结果为：

数据交换前：

```

a1 = 20      a2 = 35
b1 = 3.25    b2 = -4.86
c1 = a       c2 = b
d1 = abcdef  d2 = ghijk

```

数据交换后：

```

a1 = 35      a2 = 20
b1 = -4.86   b2 = 3.25
c1 = b       c2 = a
d1 = ghijk   d2 = abcdef

```

## 6.7 函数指针

我们知道，一个数组的数组名是一个指针常量，它指向该数组对应存储空间的开始地址，即它的值是第一个元素的存储地址。同样，一个函数的函数名也是一个指针常量，它指向该函数执行代码对应存储空间的开始位置，即它的值为保存函数执行代码的首地址。当调用一个函数时，实际上是根据该函数名找到对应执行代码的首地址，从而能够执行这段代码，即调用这个函数。因为定义一个函数的格式为：

<函数类型> <函数名> (<参数表>) {...}

则<函数名>是一个指向函数的指针，该指针类型为：

<函数类型> (\* ) (<参数表>)

该<参数表>中可以只保留参数类型，省略其参数名。如：

- (1) void f1(int x);
- (2) int f2(int a[], int n);
- (3) char \* f3(char \* a, const char \* b);
- (4) void f4(int& x, double d);

每个函数名所对应的函数指针类型分别为：

- (1) void (\* )(int)
- (2) int (\* )(int[], int)

(3) char \* ( \* )(char \* , const char \* )

(4) void ( \* )(int&, double)

若把一个指针定义为指向一种函数的指针类型,并把这种函数的函数名赋予这个指针,则这个指针就可以同函数名一样使用,出现在函数名能够出现的任何地方。如:

(1) void( \* pf1)(int) = f1;

(2) int ( \* pf2)(int[ ], int) = f2;

(3) char \* ( \* pf3)(char \* , const char \* ) = f3;

(4) void ( \* pf4)(int&, double) = f4;

以后使用 pf1, pf2, pf3 和 pf4 就如同使用 f1, f2, f3 和 f4 一样。如函数调用 pf1(25)同函数调用 f1(25)完全一样。

在一个函数定义的参数表中,每个参数的类型可以是普通、指针、引用和数组类型,除此之外还可以是函数类型。同数组类型实际上是指向元素的指针类型一样,函数类型实际上也是指向函数的指针类型。如假定一个函数参数说明为 void f(int),则等价于指向该函数的指针说明 void( \* f)(int)。

对于一个函数参数或函数指针参数,它的作用域同其他形参一样,都是这个函数的函数体。函数参数或函数指针参数所对应的实参必须是同种类型函数的函数名或函数指针。当进行实虚结合时,将把作为实参的函数名或函数指针的值传送给对应形参所占用的存储空间中,当在带有函数参数的函数体中使用函数形参进行函数调用时,实际上是调用对应实参所表示的函数。

下面是使用函数参数的一个程序的例子。

```
#include <iostream.h>
void swop(int& x, int& y) // 交换 x 和 y 的值
{
    int w = x; x = y; y = w;
}
void selectMax(int a[], int n1, int n2, int& k)
    // 从 a[n1]至 a[n2]中顺序查找出具有最大值的元素,
    // 将该元素的下标赋给 k 带回
{
    k = n1;
    for(int i = n1 + 1; i <= n2; i++)
        if(a[i] > a[k]) k = i;
}
void selectSort(int a[], int n, void f1(int&, int&))
    // 对数组 a[n]按降序进行选择排序
{
    for(int i = 0; i <= n - 2; i++) {
        int k;
        selectMax(a, i, n - 1, k); // 从一趟区间中查找最大值
        f1(a[i], a[k]); // 交换 a[i]和 a[k]元素的值
    }
}
void main()
{
```

```

int a[8] = {34,25,68,50,76,13,45,64};
selectSort(a,8,swop);
for(int i=0;i<8;i++) cout << a[i] << ' ';
cout << endl;
}

```

该程序的运行结果为:

76 68 64 50 45 34 25 13

## 习题六

### (一) 填空题

1. 一个函数调用表达式能够作为左值的条件是:函数的返回值必须是\_\_\_\_\_类型。
2. 假定一个函数的数组参数说明为 `char a[]`,则等价的指针参数说明为\_\_\_\_\_。
3. 假定一个函数的二维数组参数说明为 `int w[][N]`,则等价的指针参数说明为\_\_\_\_\_。
4. 假定一个参数说明为 `const int a`,则在函数体中\_\_\_\_\_ (能够/不能够)改变 `a` 的值。
5. 假定一个参数说明为 `const char * p`,则在函数体中\_\_\_\_\_ (能够/不能够)改变 `p` 所指向的存储空间的内容,但\_\_\_\_\_ 改变 `p` 指针的内容。
6. 假定一个参数说明为 `int& x`,则进行函数调用时,它是对应\_\_\_\_\_ 的别名。
7. 一个函数带有函数声明时,则参数的默认值应该在\_\_\_\_\_ (函数定义/函数声明)中给出。
8. 在进行函数调用时,将把\_\_\_\_\_ 传递给非引用参数,把实参的\_\_\_\_\_ 传递给引用参数。
9. 函数执行中对引用参数的访问实际上就是对相应\_\_\_\_\_ 的访问。
10. 当函数的返回类型为 `void` 时,在函数体中可以使用\_\_\_\_\_ 语句,否则必须使用\_\_\_\_\_ 语句返回一个值。
11. 当实参为一个数组名时,对应的形参必须是\_\_\_\_\_ 类型或\_\_\_\_\_ 类型。
12. 变量的四种作用域分别为\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和\_\_\_\_\_。
13. 不同作用域范围内的变量\_\_\_\_\_ (能够/不能够)同名。
14. 当一个函数的非引用参数为 `a`,假定它对应的实参也为 `a`,则在函数体中对 `a` 的访问与对应的实参 `a` \_\_\_\_\_ (有关/无关)。
15. 当一个函数的引用参数为 `a`,假定它对应的实参也为 `a`,则在函数体中对 `a` 的访问与对应的实参 `a` \_\_\_\_\_ (有关/无关)。这两个变量的作用域\_\_\_\_\_ (相同/不同)。
16. 当一个函数的引用参数为 `a`,假定它对应的实参为 `b`,则它们对应的作用域\_\_\_\_\_ (相同/不同),访问它们的存储空间\_\_\_\_\_ (相同/不同)。
17. 在函数定义的第一层复合语句内\_\_\_\_\_ (能够/不能够)定义与形参同名的变



量,因为它们的作用域\_\_\_\_\_ (相同/不同)。

18. 函数的形参和函数体内的变量\_\_\_\_\_ (可以/不可以)与全局域或文件域的对象同名。

19. 在两个嵌套的不同作用域内若分别定义有同名对象,当在内层作用域访问这个对象时,将访问的是在\_\_\_\_\_ (内层/外层)定义的对象,除此之外将访问的是在\_\_\_\_\_ (内层/外层)定义的对象。

20. 在一个程序文件中定义的函数,若要在另一个程序文件中访问,则必须在该文件开始给出一条该函数的\_\_\_\_\_ 对其进行说明。

21. 在一个函数体中可以使用函数调用表达式调用其他函数,也可以调用\_\_\_\_\_。

22. 在函数体中又调用自身函数称为\_\_\_\_\_调用,该函数称为\_\_\_\_\_函数。

23. 当两个函数的函数名\_\_\_\_\_,但参数的个数或对应参数的类型\_\_\_\_\_时,则称为重载函数。

24. 当一个函数为 `void f(int, char = 'a')`,另一个函数为 `void f(int)`,则它们\_\_\_\_\_ (是/不是)重载函数,在一个程序中\_\_\_\_\_ (可以/不可以)同时定义这两个函数。

25. 一个函数模板中的类型参数所对应的具体类型由调用该函数模板的\_\_\_\_\_决定。

26. 当在同一个程序中存在一个普通函数是一个函数模板的重载函数时,则与函数调用表达式相符合的\_\_\_\_\_将被优先调用执行。

27. 当一个函数调用表达式只能与一个函数模板相符合时,将首先根据函数模板生成一个\_\_\_\_\_,然后再调用它执行。

28. 在函数定义的参数表中,也可以使用函数参数,假定一个函数参数说明为 `int& f(int [], int)`,则等价的函数指针说明为\_\_\_\_\_。

## (二) 给出下列程序运行后的输出结果并上机验证

```
1. #include <iostream.h>
void main() {
    int a=10, b=20;
    cout << a << ' ' << b << endl;
    { a*=3;
      int b=a+35;
      cout << a << ' ' << b << endl;
    }
    cout << a << ' ' << b << endl;
}
```

```
2. #include <iostream.h>
int a=5;
void main() {
    int a=10, b=20;
    cout << a << ' ' << b << endl;
    { int a=0, b=0;
      for(int i=1; i<6; i++) {
          a+=i; b+=a;
      }
    }
}
```

```

    }
    cout << a << ' ' << b << ' ' << ::a << endl;
}
cout << a << ' ' << b << endl;
}

```

```

3. #include <iostream.h>
int f1(int x, int y)
{
    x=x+y; y=x+y;
    cout << "x=" << x << ", y=" << y << endl;
    return x+y;
}
void main()
{
    int x=5,y=8;
    int z=f1(x,y);
    cout << "x=" << x << ", y=" << y;
    cout << ", z=" << z << endl;
}

```

```

4. #include <iostream.h>
void f2(int& x, int& y)
{
    int z=x; x=y; y=z;
}
void f3(int * x, int * y)
{
    int z= *x; *x= *y; *y= z;
}
void main()
{
    int x,y;
    x=10; y=26;
    cout << "x,y=" << x << ", " << y << endl;
    f2(x,y);
    cout << "x,y=" << x << ", " << y << endl;
    f3(&x,&y);
    cout << "x,y=" << x << ", " << y << endl;
    x++; y--;
    f2(y,x);
    cout << "x,y=" << x << ", " << y << endl;
}

```

```

5. #include <iostream.h>
void f4(int a[], int n, int& s)
{
    s=0;
    for(int i=0; i<n; i+=2) s+=a[i];
}
void main()

```

```

    {
        int a[5] = {2,7,5,4,9};
        int b[10] = {4,8,6,9,2,10,7,12,6,15};
        int x;
        f4(a,5,x);
        int y = x;
        f4(b,8,x);
        y += x;
        f4(b+3,5,x);
        cout << x+y << endl;
    }
}

6. #include<iostream.h>
int x = 5;
void f5(int a);
int f6(int x);
void main()
{
    int x = 3;
    f5(x);
    f5(x+4);
    cout << x+::x << endl;
}

void f5(int a)
{
    a += x;
    cout << f6(a) << endl;
}

int f6(int x)
{
    x * = 3;
    return x+1;
}

7. #include<iostream.h>
#include<string.h>
char * f7(char * &a, int n)
{
    a = new char[n];
    strcpy(a,"motion");
    char * b = new char[n];
    strcpy(b,"telephone");
    return b;
}

void main()
{
    char *p1;
    char * p2 = f7(p1,12);
    cout << p1 << ' ' << p2 << endl;
    delete p1; delete p2;
}

```

```

8. #include<iostream.h>
const N=15;
int f8(char a[] (N), int m)
{
    int c=0;
    for(int i=0; i<m; i++) {
        int j=0;
        while(a[i][j]) {
            if(a[i][j] >= '0' && a[i][j] <= '9') c++;
            j++;
        }
    }
    return c;
}
void main()
{
    char b[4][N] = {"12ab3", "70542", "abc25", "x+y=26"};
    int c1 = f8(b, 4);
    int c2 = f8(b+2, 2);
    cout << c1 << ' ' << c2 << endl;
}

```

```

9. #include<iostream.h>
int f9(int x)
{
    cout << x << ' ';
    if(x <= 0) {cout << endl; return 0;}
    else return x*x + f9(x-1);
}
void main()
{
    int x = f9(6);
    cout << x << endl;
}

```

```

10. #include<iostream.h>
const N1=8, N2=6;
int average(int a[], int n)
{
    int s=0;
    for(int i=0; i<n; i++) s += a[i];
    return s/n;
}
double average(double a[], int n)
{
    double s=0;
    for(int i=0; i<n; i++) s += a[i];
    return s/n;
}
void main()
{

```

```

    int a[N1] = {3,6,5,10,8,2};
    double b[N2] = {3.2, 5, 6.2, 5.6, 4.9, 8.4};
    int v1; double v2;
    v1 = average(a,N1);
    v2 = average(b,N2);
    cout << "v1 = " << v1 << endl;
    cout << "v2 = " << v2 << endl;
}

```

```

11. #include <iomanip.h>
const N=10;
template< class DataType>
bool insert(DataType a[], int& n, DataType x);
template< class DataType>
void print(DataType a[], int& n);
void main()
{
    int a1[N] = {25,48,50,82,66,43};
    char a2[N] = "student";
    char * a3[N] = {"File","Edit","Insert","Project"};
    int b1=6,b2=7,b3=4;
    int n=75; char ch='w'; char * p="Build";
    insert(a1,b1,n);
    insert(a2,b2,ch);
    insert(a3,b3,p);
    print(a1,b1);
    print(a2,b2);
    print(a3,b3);
}

template< class DataType>
bool insert(DataType a[], int& n, DataType x)
{
    if(n<0) {cout << "操作失败!" << endl; return false;}
    a[n] = x; n++;
    cout << "操作成功!" << endl;
    return true;
}

template< class DataType>
void print(DataType a[], int& n)
{
    for(int i=0;i<n;i++) {
        cout << setw(10) << a[i];
        if((i+1)%4 == 0) cout << endl;
    }
    if(i%4) cout << endl;
}

```

```

12. #include <iostream.h>
void fun10(int * a[], int m, int n)
{

```

```

    int i, j;
    for(i = 0; i < m; i++)
        a[i] = new int[n];
    for(i = 0; i < m; i++)
        for(j = 0; j < n; j++)
            a[i][j] = (i + 1) * (j + 1);
}

void main()
{
    int m1, n1;
    cout << "输入矩阵的行数和列数:";
    cin >> m1 >> n1; // 假定输入 3 和 4
    int ** b = new int * [m1];
    fun10(b, m1, n1);
    for(int i = 0; i < m1; i++) {
        for(int j = 0; j < n1; j++)
            cout << b[i][j] << ' ';
        cout << endl;
    }
}

```

### (三) 指出下列每个函数的功能

1. `int fun1(int n)`

```

{
    int p = 1, s = 0;
    for(int i = 1; i <= n; i++) {
        p * = i; s += p;
    }
    return s;
}

```
2. `#include <iostream.h>`  
`#include <stdlib.h>`  
`#include <math.h>`  
`double fun2(double x, double y, char op)`

```

{
    switch(op) {
        case '+': return x + y;
        case '-': return x - y;
        case '*': return x * y;
        case '/':
            if(fabs(y) < 1.0E-6) {
                cout << "除数为 0, 退出运行!" << endl;
                exit(1);
            }
            return x / y;
        default:
            cout << "运算符非法!" << endl;
            exit(1);
    }
}

```

```

    }

3. void fun3(int a[], int n)
{
    int i, j, x;
    for(i = 1; i < n; i++) {
        x = a[i];
        for(j = i - 1; j >= 0; j--)
            if(x < a[j]) a[j + 1] = a[j];
            else break;
        a[j + 1] = x;
    }
}

4. void fun4(double& x, double& w, int n)
{
    cout << "输入一个实数:";
    cin >> x;
    double y = 1;
    w = 0;
    for(int i = 1; i <= n; i++) {
        y * = x; w += y/i;
    }
}

5. void fun5(char* a, const char* b)
{
    while(*b) *a++ = *b++;
    *a = 0;
}

6. int fun6(int n)
{
    if(n == 0) return 1;
    else return 2 * fun6(n - 1);
}

7. template< class DT>
int fun7(DT& x, DT& y)
{
    if(x > y) return 1;
    else if(x == y) return 0;
    else return -1;
}

8. template< class T>
bool fun8(T a[], int n, T key)
{
    for(int i = 0; i < n; i++)
        if(a[i] == key) return true;
    return false;
}

```

```

9. #include <stdlib.h>
   #include <time.h>
   void fun10(int * &a, int n)
   {
       srand(time(0));
       int i;
       a = new int[n];
       for(i = 0; i < n; i++)
           a[i] = rand() % 100;
   }

10. void fun10(int * * &a, int m, int n)
    {
        int i, j;
        a = new int * [m];
        for(i = 0; i < m; i++)
            a[i] = new int[n];
        cout << "输入" << m << " * " << n << " 整数矩阵" << endl;
        for(i = 0; i < m; i++)
            for(j = 0; j < n; j++)
                cin >> a[i][j];
    }

```

#### (四) 编写下列程序

1. 编写一个函数,求出一维整型数组  $a[n]$  中所有元素的平方之和。

```
int fun1(int a[], int n);
```

2. 编写一个函数,分别求出一维整型数组  $a[n]$  中所有奇数元素的个数和所有偶数元素的个数。

```
void fun2(int a[], int n, int& c1, int& c2);
```

3. 编写一个函数,从一个二维整型数组中查找具有最大值的元素,由引用参数 row 和 col 带回该元素的行号和列号。

```
void fun3(int a[][N], int m, int& row, int& col); // N 为常量
```

4. 编写一个函数,求出由指针 a 所指向的字符串中包含的每种十进制数字出现的次数,把统计结果保存在由指针 b 所指向的整型数组中。

```
void fun4(char* a, int* b);
```

5. 编写一个递归函数过程,求出两个自然数 m 和 n 的最大公约数。

```
int fun5(int m, int n);
```

6. 编写一个递归函数过程,求出两个自然数 m 和 n 的最小公倍数。

```
int fun6(int m, int n, int b=2);
```

7. 编写一个程序,求出二元一次方程组 
$$\begin{cases} a_{11}x + a_{12}y = a_{13} \\ a_{21}x + a_{22}y = a_{23} \end{cases}$$
 的解,其中方程组的系数用一个



实数二维数组保存。要求编写出一个主函数和两个普通函数,一个普通函数用于从键盘上向数组输入数据,另一个普通函数用于求出以该数组为系数矩阵的对应方程组的解,并由引用参数 x 和 y 返回所求的两个根,还有当方程组有惟一解时返回真,否则返回假。程序中的主函数用来定义一个二维实型数组,依次调用这两个普通函数,并且输出所求得解。

提示:方程的两个根 x0 和 y0 分别为:

$$x_0 = \frac{a_{13}a_{22} - a_{12}a_{23}}{a_{11}a_{22} - a_{12}a_{21}} \quad y_0 = \frac{a_{11}a_{23} - a_{13}a_{21}}{a_{11}a_{22} - a_{12}a_{21}}$$

当  $a_{11}a_{22} - a_{12}a_{21} \neq 0$  时有惟一解。

## 第七章 结构与联合

结构(struct)是一种用户自定义的类型。我们已经学习过的所有类型,即整型(int)、字符型(char)、浮点型(float、double)、逻辑型(bool)以及相应的指针类型、引用类型和数组类型等,它们都是C++系统中内定义的数据类型(又称为标准类型或预定义类型),系统为它们规定了相应的取值范围和操作(运算),在程序中可以直接使用。如系统规定整型的取值范围是 $-2^{31} \sim +2^{31} - 1$ 之间的所有整数,所能进行的运算包括各种算术运算、比较运算、逻辑运算和赋值运算等,当然这些运算的具体对象是整型常量和变量。对于用户自定义的数据类型(简称用户类型或自定义类型),它包含数据和操作两个部分,数据部分由已有类型(包括预定义类型和已经定义的用户类型)的变量所组成,操作部分由对数据部分进行各种操作的函数所组成。用户类型中定义的每个变量称为数据成员,每个函数称为函数成员或成员函数。无论是数据成员还是成员函数统称为该类型的成员。用户自定义类型除了已经提到的结构(struct)之外,还有联合(union)和类(class)。通常在使用结构和联合时只定义它的数据成员,不定义它的成员函数,对其数据成员的操作是通过调用外部函数(即在该类型之外定义的函数)实现的;而在类类型中通常既定义数据成员又定义成员函数。本章和下一章将陆续介绍结构、联合和类类型的定义和使用。

### 7.1 结构的定义

C++语言中的预定义数据类型只能用来描述简单数据,如可用整型描述人的年龄,用字符串型(即字符指针或字符数组型)描述人的姓名,用浮点型描述人的工资等。但对于复杂的数据,即包含有一个或多个数据项,各数据项可以具有相同或不同的类型,并且每个数据项的含义不同,这就无法由预定义类型进行整体描述,必须由用户定义的类型来描述。如要描述一个人的记录数据,假定它包含姓名、性别、年龄和工资这四个数据项,则可以使用一种结构类型来描述。设该结构类型的名字用标识符 Person 表示,其中的姓名数据项用标识符 name 表示,对应类型为字符串型;性别数据项用标识符 sex 表示,对应类型为布尔(逻辑)型,假定分别用布尔常量 true 和 false 表示男和女;年龄数据项用标识符 age 表示,对应类型为整型;工资数据项用标识符 pay 表示,对应类型为浮点型。整个 Person 结构类型可定义为:

```
struct Person {  
    char name[10];  
    bool sex;  
    int age;  
    float pay;  
};
```

用此 Person 类型定义的每一个变量可以具体表示(存储)一个人的记录,该变量的 name、sex、age 和 pay 域(成员)用来分别存储一个人的姓名、性别、年龄和工资。

### 7.1.1 结构定义格式

上面定义的 Person 结构类型是一个具体的例子。在 C++ 中,结构类型的定义格式如下:

```
struct <结构类型名> {  
    <成员类型名 1> <成员名 1>;  
    <成员类型名 2> <成员名 2>;  
    ⋮  
    <成员类型名 n> <成员名 n>;  
};
```

一个结构类型的定义以关键字 struct 开始,后跟一个作为结构类型名的标识符,然后从左花括号之后进行成员定义,以右花括号结束成员定义,左右花括号之间称为结构体,最后以分号结束整个结构的定义。

结构定义中的 <结构类型名> 为用户命名的任何一个有效的标识符,以后使用它就如同使用任何一种简单类型名一样,允许出现在简单类型名能够出现的任何地方,利用它能够定义具有该结构类型的变量或函数;<成员类型名 1> ~ <成员类型名 n> 用来定义该结构所包含的成员类型,每个成员类型名必须是一种已有的类型,<成员名 1> ~ <成员名 n> 为 n 个由用户命名的有效的标识符,用它们表示该类型所含的 n 个数据成员,每一个数据成员分属于相应的类型,当若干个成员名具有同一种成员类型时,可将它们定义在同一种成员类型名之后,各成员名之间用逗号分开,当然在结束同一类型的成员定义后要用分号结束定义。

### 7.1.2 定义格式举例

```
(1) struct A {  
    int a,b,c;  
};  
  
(2) struct B {  
    char ch;  
    int x,y;  
    double z;  
};  
  
(3) struct C {  
    char *cp;  
    int a[5];  
};  
  
(4) struct D {  
    int *a;
```

```

        int *ap;
        int maxsize;
    };

(5) struct E {
        int d, *e;
        B b;
    };

(6) struct F {
        double data;
        F *next;
    };

```

上述定义的(结构)类型 A 包含有三个整型成员 a, b, c; 类型 B 包含有一个字符型成员 ch, 两个整型成员 x 和 y, 以及一个双精度浮点型成员 z; 类型 C 包含有一个字符指针型成员 cp 和一个具有 5 个元素的整型数组 a; 类型 D 包含有两个整数指针型成员 a 和 ap, 以及一个整型成员 maxsize; 类型 E 包含有一个整型成员 d, 一个整数指针型成员 e 和一个 B 结构类型成员 b; 类型 F 包含有一个双精度浮点型成员 data 和一个 F 结构型指针成员 next。

### 7.1.3 结构使用说明

1. 在一个结构的定义中, 其成员类型可以是除本身结构类型之外的任何已有类型, 也可以是任何已有类型(包括本身类型在内)的指针类型(数组可看做常量指针类型)。如在上面的格式举例(6)中, next 为指向本身结构的指针成员, 这是允许的, 但若把 next 定义为 F 的直接成员, 则是非法的, 因为这种递归嵌套定义, 将无法确定它的对象所需占用的存储空间的大小。

2. 当一个结构类型定义在函数之外时, 它具有文件作用域, 若定义在任一对花括号之内, 则具有局部作用域, 其作用域范围是所在花括号构成的块。当然使用数据类型也同使用数据对象一样, 必须遵循先定义后使用的原则, 即只有被定义了一种数据类型后, 才能够用它来定义变量、定义函数参数或作为函数的返回类型。

3. 在程序中同一个作用域内用户类型名是惟一的, 即不允许出现重复的类型标识符或其他同名量, 但在不同的作用域内用户类型名可以重复, 它们不会发生冲突。如假定类型 A 为文件作用域, 但在一个函数中又用标识符 A 定义了另一个类型, 则这个类型 A 的作用域局限在这个函数中, 文件作用域类型 A 在这个函数内被局部类型 A 所取代, 而在该函数之外的所有地方起作用。

4. 每个结构类型定义中的成员名在该类型中必须惟一, 但在整个程序中不要求惟一, 它可以同程序中的类型名(包括本身类型)、变量名、函数名以及任何类型中的成员名重名, 都是允许的。因为当引用一种类型中的成员时, 总是与它所属的对象(变量)联系起来, 对象名成了成员名的限定词, 所以不会与其他同名量产生二义性。如 a, x.a, y.a 各不相同, a 表示一个独立变量, x.a 表示 x 对象中的成员 a, y.a 表示 y 对象中的成员 a, 它们都是惟一的, 无二义性。

5. 若在定义一个结构类型 AA 时需要使用另一个结构类型 BB 作为其成员类型, 而定义

BB 时又需要使用 AA 作为其成员类型,这就使它们的定义互为先决条件。在 C++ 中如何处理这种特殊情况呢?由此引入了不完整定义的概念。一个用户类型的不完整定义是指只给出它的类型关键字和类型标识符而不给出定义体就结束定义的情况。不完整定义的类型只能作为指针类型使用,并且必须在稍后给出它的完整定义。例如:

```
struct BB;      // BB 类型的不完整定义
struct AA {     // AA 类型的完整定义
    char a;
    BB * b;     // 允许使用 BB 的指针类型
};
struct BB {     // 给出 BB 的完整定义
    int b;
    AA a;       // 使用了刚定义的 AA 类型
};
```

6. 类型定义语句属于非执行语句,只在程序编译阶段处理它,并不在编译后生成的目标程序中存在对应的可执行目标代码。进行编译处理时,是把类型名称和分类(即结构、联合或类),所含成员的每个成员名称、类型及大小(即该类型的数据所占存储空间的字节数),整个类型的大小(它等于各成员大小之和)及类型作用域等信息登记到系统中,待以后定义该类型的变量时使用。

## 7.2 结构变量的定义和初始化

一种结构类型定义后,就可以利用它在其作用域内定义变量并进行必要的初始化,这如同利用标准类型定义变量并进行初始化的情况一样。每定义一个变量,系统就按照所属类型的大小为其分配相应的存储空间,若定义中包含有初始化数据,则求出其值并赋给该变量的存储空间中,以后对变量的访问就是对相应的存储空间存取信息。

对结构变量的定义可采用以下三种格式。

### 7.2.1 用结构类型名定义变量

具体定义格式为:

```
[struct] <结构类型名> <变量名> [= (<初始化数据>)] | <同类型变量名>],...;
```

它就是以前介绍过的变量定义语句,只是把标准类型关键字替换为用户类型名而已。在此格式中,用中括号表示其中的内容为任选项,竖线表示其前后两项任选其一。语句格式中的 struct 关键字可写可不写,不影响语句功能,<结构类型名>是已定义的结构类型,<变量名>是由用户命名的任何有效的标识符,用它表示一个结构变量,变量名后的中括号内为初始数据项,若需要对变量进行初始化则使用它,否则省略它。用户给出的初始数据项可以用花括号括起来的由每一个成员值(成员值之间用逗号分开)构成的<初始化数据>,也可以是同类型的另一个变量。利用同类型变量初始化就是将其值拷贝到被定义的变量中,利用初始化数据对结构变量进行初始化就是将其的每一个成员值依次拷贝到变量

的相应域中。初始化数据中的成员值个数可以小于变量的成员数,在这种情况下,结构变量中后面未被初始化的成员由系统自动置为 0。

同一般的变量定义语句一样,在此语句中既可以定义结构变量,也可以定义结构指针变量、结构数组和结构指针数组,并且每一种都可以定义任意多个,但每个变量定义之间要用逗号分开,最后以分号结束整个语句。

同简单变量一样,当全局或静态(static)结构变量未被初始化时,它的每个成员被系统自动置为 0,当自动(auto)结构变量(即不带 static 保留字定义的结构变量)未被初始化时,它的每个成员的值是随意的,即不确定的。

格式举例:

假定一个结构类型 Arith 包含有一个字符成员 op 与两个整数成员 a 和 b,则该类型定义为:

```
struct Arith {
    char op;
    int a,b;
};
```

因每个字符占一个字节,每个整数(int 型)占一个机器字长,即 4 个字节,所以 Arith 类型大小应该为 9,但系统通常为一个结构对象分配整数倍大小的机器字长,所以 Arith 类型的大小实际上为 12 而不是 9,此时 op 成员也同样占有 4 个字节,其中只有第一个字节有用,其后三个字节未用。

又假定有如下一条对整型变量 xx 的定义语句:

```
int xx = 40;
```

下面每一条结构变量定义语句都是正确的。

- (1) Arith x,y;
- (2) Arith z1 = {'+',10,xx}, z2 = {'\*',60}, z3 = z1;
- (3) Arith \*d = &z1;
- (4) Arith a[4] = {'+',3,7}, {'-',10,5}, {'\*',6,4}, {'/',8,5};
- (5) Arith \*b[] = {&z1,&z2,a+2,&a[3]};

在上述语句(1)中,定义了类型为 Arith 的两个结构变量 x 和 y;语句(2)定义了三个结构变量 z1,z2 和 z3,并分别对它们进行了初始化,使得 z1 的 op,a 和 b 成员的值分别为字符 '+',整数 10 和 xx 的值 40,z2 的成员值依次为 '\*'、60 和 0,z3 被初始化为 z1 的值,其成员值同样为 '+',10 和 40;语句(3)定义了结构指针,即指向 Arith 结构类型的指针变量 d,并用 z1 的地址来初始化,使其指向变量 z1;语句(4)定义了一个 Arith 类型的结构数组 a,它包含有 4 个元素,并被依次初始化,如 a[2]元素中的成员被初始化 '\*'、6 和 4;语句(5)定义了一个 Arith 结构指针数组 b,其元素个数等于初始化表中所列指针的个数 4,每个元素的初值依次为结构变量 z1 的地址、结构变量 z2 的地址、数组 a 中 a[2]元素的地址和 a[3]元素的地址。

当系统执行语句(1)时,将为 x 和 y 分配大小均为 12 的存储空间,执行语句(2)时也同样将为 z1,z2 和 z3 分配大小均为 12 的存储空间,执行语句(3)时为 d 分配一个字即 4 字节的存储空间,执行语句(4)和(5)时,分别为数组 a 和 b 分配 48 和 16 个字节的存储空间。

我们知道利用 new 运算符能够创建动态变量和动态数组,利用 delete 运算符能够删除动态变量和动态数组,即删除它们所占有的动态存储空间,同样,利用它们也能够创建或删除动态结构变量和动态结构数组,但创建时不能够对其进行初始化。通过 new 运算符创建一个动态结构变量或动态结构数组后返回的同样是其对应的存储空间的首地址,把这个首地址赋给一个同类型的结构指针后,就可以利用这个指针访问所指向的动态结构变量或数组。例如:

```
Arith *p = new Arith;  
Arith *a = new Arith[n];
```

第一条语句创建了一个具有 Arith 结构的动态变量,并将它的地址赋给指针 p,第二条语句创建了一个含有 n 个元素的具有 Arith 结构的动态数组,并将数组的首地址赋给指针 a。

当不需要动态变量或动态数组时,必须使用 delete 运算符把它删除,释放所占有的存储空间,否则将一直占有着,直到程序运行结束为止。例如:

```
delete p;  
delete []a;
```

将删除 p 指针所指向的动态变量和 a 指针所指向的动态数组。

而对于非动态分配的结构变量或结构数组,同一般变量或数组一样,当离开作用域后,所占用的存储空间将自动被系统收回。

### 7.2.2 定义结构类型的同时定义变量

具体定义格式为:

```
struct <结构类型名> {  
    <成员类型名 1> <成员名 1>;  
    <成员类型名 2> <成员名 2>;  
    :  
    <成员类型名 n> <成员名 n>;  
} <变量名> [= (<初始化数据>) | <同类变量名>], ...;
```

格式举例:

```
struct AAA {  
    char s[20];  
    int top;  
} a1 = {"Microsoft", 0}, a2 = a1, a3, *ap;
```

此语句在定义 AAA 结构类型的同时定义三个变量 a1, a2 和 a3 以及一个指针变量 ap, 其中 a1 被初始化为 {"Microsoft", 0}, a2 被初始化为 a1, a3 和指针 ap 未被初始化。

### 7.2.3 定义无名结构类型的同时定义变量

具体定义格式为:

```
struct {
```

```

    <成员类型名 1> <成员名 1>;
    <成员类型名 2> <成员名 2>;
    :
    :
    <成员类型名 n> <成员名 n>;
} <变量名> [= {<初始化数据>} | <同类变量名>], ...;

```

这种格式由于没有类型名,所以只能随时定义变量,以后无法利用它定义变量或函数,因此较少使用它。下面是将它用在用户类型中对成员定义的情况。

```

struct BBB {
    char name[10];
    struct { // 无名结构
        int yy, mm, dd; // 无名结构体
    } birth; // 无名结构变量,它含有三个整数域 yy, mm 和 dd
} bx = {"xxk", {55, 3, 27}};

```

BBB 结构含有两个成员 name 和 birth, name 为具有 10 个元素的字符数组, birth 为含有三个整数域 yy, mm 和 dd 的结构变量。上述在定义 BBB 结构的同时定义了 bx 变量,并对它进行了初始化,使 name 成员保存的值为"xxk", birth 成员保存的值为{55, 3, 27}, 其中 yy 的值为 55, mm 的值为 3, dd 的值为 27。

### 7.3 结构成员的访问操作

定义结构变量之后就可以利用它存取具体结构数据,系统对结构变量所提供的运算有赋值(=)、直接指定成员(.)和间接指定成员(->)三种,这三种运算符(即=、. 和->)分别称为赋值运算符、直接成员运算符(又称点运算符)和间接成员运算符(又称箭头运算符)。它们都是双目运算符,并且成员运算符同下标运算符和函数运算符一样具有最高的优先级,而赋值运算符的优先级较低。

赋值运算符的两边为同类型的结构变量,即为同一结构类型标识符所定义的变量,运算功能是把右边变量的值拷贝到左边变量中,即拷贝到左边变量所对应的存储空间中,运算的结果为左边的变量。赋值号可以连续使用,并且规定结合性为从右到左,所以若 z1, z2 和 z3 为同类型的结构变量,则赋值语句 z3 = z2 = z1 的执行过程是首先把 z1 赋给 z2,再接着把 z2 赋给 z3,使得 z3 和 z2 都具有 z1 的值。

直接成员运算符的左边是一个结构变量(包括结构数组中的元素),右边是该结构变量中的一个成员,运算结果是一个结构(变量)中的成员变量。如 x.a 表示 x 中的成员变量 a; x.b.t 表示 x 中 b 成员内的成员变量 t, 其中 b 又是 x 中的结构成员; vec[5].name 表示结构数组 vec 中第 5 号元素内的成员变量 name。

间接成员运算符的左边是一个结构指针变量,右边是该结构指针变量所指结构中的一个成员,运算结果是一个指针所指结构中的一个成员变量。如 p -> a 表示 p 指针所指向结构中的成员变量 a, 它可以等价表示为 (\*p).a, 其中括号内为 p 指针所指的结构变量,此处用圆括号括起来是必须的,若写成 \*p.a 则是错误的,因为成员运算符的优先级高于取内容运算符的优先级,这样先做的是点运算,而不是星号运算; p -> c -> n 表示 p 指针所指结构



中的指针成员 c,再接着得到由 c 所指结构中的成员变量 n,它可以等价表示为(\*p).c->n,(\*(\*p).c).n 或(\*p->c).n;list[n]->wage 表示结构指针数组 list 中第 n 号元素所指结构中的成员变量 wage。

C++ 中的其他运算符,如算术运算符和关系运算符等,只有通过以后学习的运算符重载函数定义后才能够应用到结构变量上,否则是不能施加于结构类型的变量的。

通过成员运算符(直接或间接)能够得到结构中的成员变量,每个成员变量与相同类型的简单变量或数组元素一样,能够作为左值或右值参与该类型所具有的各种运算。

## 7.4 使用结构的程序举例

例 1. 用结构数组保存数据。

```
#include<iostream.h>
#include<string.h>
struct Person { //定义结构类型 Person
    char name[10]; //姓名
    bool sex;      //性别
    int age;       //年龄
    float pay;     //工资
};
Person a[10]; //定义全局结构数组 a,大小由用户决定,这里假定为 10
void input(int n)
    //向全局结构数组 a 中输入 n 个记录
{
    cout << "从键盘上输入具有 Person 结构的" << n << "个记录:" << endl;
    int i, k;
    Person x; //定义局部结构变量 x
    for(i=0; i<n; i++) {
        cin >> x.name; //输入一个人的名字
        cin >> k; //因 C++ 没有提供为逻辑变量直接输入数据的功能,
        // 所以在此用输入 1 表示男, 0 (广义为非 1) 表示女
        if(k == 1) x.sex = true; else x.sex = false;
        // 此处含有向成员赋值的操作
        cin >> x.age >> x.pay; //输入年龄和工资
        a[i] = x; //将 x 赋给 a[i] 元素,此为结构赋值
    }
}

void output(int n)
    //显示出全局结构数组 a 中的 n 个记录
{
    cout << "显示具有 Person 结构的" << n << "个记录:" << endl;
    for(int i=0; i<n; i++) {
        cout << a[i].name << " "; //显示姓名
        if(a[i].sex == true) //显示性别,此条件可改为(a[i].sex)
            cout << "male" << " ";
        else
            cout << "female" << " ";
    }
}
```

```

        cout << "female" << ' ';
        cout << a[i].age << ' ' << a[i].pay << endl; // 显示年龄和工资
    }
}

void main()
{
    int n;
    cout << "请输入一个正整数(1 <= n <= 10):";
    cin >> n;
    input(n);
    output(n);
}

```

假定程序运行后从键盘上输入数值 3 到变量 n 中,则程序输入和运行结果如下:

```

请输入一个正整数(1 <= n <= 10):3
从键盘上输入具有 Person 结构的 3 个记录:
xxk 1 45 460
hexx 1 50 440
wchf 0 44 450.5
显示具有 Person 结构的 3 个记录:
xxk male 45 460
hexx male 50 440
wchf female 44 450.5

```

例 2. 从结构数组中查找某个域的值最大的记录。

```

#include <iostream.h>
#include <string.h>
struct Person { // 定义结构 Person
    char name[10];
    bool sex;
    int age;
    float pay;
};
Person a[5] = {{"luyx", 1, 42, 386}, {"gcyng", 0, 45, 482},
               {"lumng", 1, 40, 420}, {"ningch", 1, 36, 530},
               {"wchf", 0, 46, 475}}; // 定义全局结构数组 a 并初始化

void output(int n)
    // 显示出全局结构数组 a 中的 n 个记录
{
    // 函数体同上一程序
}

void find(int n)
    // 从全局结构数组 a 的前 n 个记录中查找并显示出
    // 具有最大工资值的记录
{
    int k = 0; // 用 k 指示当前具有最大工资值元素的下标,初值为 0
    float x = a[0].pay;

```

```

        // 用 x 保存当前最大工资值,初值为 0 号元素的工资值
        for(int i=1; i<n; i++) {
            // 采用顺序比较的方法进行查找,循环结束后下标
            // 为 k 的元素具有最大工资值
            if(a[i].pay>x) {
                x=a[i].pay;
                k=i;
            }
        }
        cout << endl << "显示数组 a 中具有最大工资值的记录:" << endl;
        cout << a[k].name << ' ' << a[k].sex << ' ';
        cout << a[k].age << ' ' << a[k].pay << endl;
    }

    void main()
    {
        output(5);
        find(5);
    }

```

此程序首先输出数组 a 中的 5 个记录,然后从数组 a 的前 5 个记录中查找并显示出具有最大工资值的记录。程序运行结果如下:

显示具有 Person 结构的 5 个记录:

```

luyx male 42 386
gcyng female 45 482
luning male 40 420
ningch male 36 530
wchf female 46 475

```

显示数组 a 中具有最大工资值的记录:

```

ningch 1 36 530

```

### 例 3. 对结构数组中保存的记录进行选择排序。

分析:在给出程序之前,先讨论一下选择排序方法。假定在一个结构数组 a 中保存着 n 个记录,对应下标(编号)为 0~n-1,若要对这 n 个记录按照某个域的值(称此域为排序域,一个记录的排序域的值称为排序码)从小到大顺序排列在数组中,即使得 0 号元素 a[0] 的排序码最小,1 号元素 a[1] 的排序码仅大于等于 0 号元素的排序码,而小于等于其后所有元素的排序码,2 号元素 a[2] 的排序码仅大于等于前两个元素的排序码,而小于等于其后所有元素的排序码,……,最后一个元素 a[n-1] 的排序码最大。有许多种方法可以实现数组的排序运算,这里仅介绍一种简单的选择排序方法。该方法需要进行 n-1 趟查找和交换,第一趟从全部 n 个元素中顺序查找出排序码最小的元素,把它的值同 a[0] 元素的值相交换,使得 a[0] 中保存着排序码最小的记录,至此第一趟查找和交换结束;第二趟从 a[1]~a[n-1] 元素中顺序查找出排序码最小的元素,把它的值同 a[1] 元素的值相交换,使得 a[1] 中保存着排序码仅大于等于 a[0] 元素的排序码,而小于等于其后所有元素排序码的记录,至此第二趟查找和交换结束;依次类推,当进行 n-1 趟时,前面 n-2 个元素都已有序,需从 a[n-2] 和 a[n-1] 元素中顺序查找出排序码最小的元素,然后把它的值同 a[n-2] 元素的值相

交换,至此最后一趟结束,亦即整个排序过程结束,此时数组  $a$  中的记录就按照排序码从小到大的顺序排列了。

假定数组  $a$  中保存着 5 个记录,每个记录的排序码依次为 48,35,64,27 和 44,每个记录的其余部分若用  $R_i$  表示, $i$  为记录序号,则数组  $a$  中的内容如下:

0	1	2	3	4
$R_0$ 48	$R_1$ 35	$R_2$ 64	$R_3$ 27	$R_4$ 44

进行第一趟时从全部 5 个元素中顺序查找出具有最小排序码的元素  $a[3]$ ,把它的值与  $a[0]$  元素的值交换后,数组  $a$  的当前内容为:

0	1	2	3	4
$R_3$ 27	$R_1$ 35	$R_2$ 64	$R_0$ 48	$R_4$ 44

进行第二趟时从后面 4 个元素中顺序查找出具有最小排序码的元素  $a[1]$ ,把它的值与  $a[1]$  元素的值交换后,数组  $a$  的状态不变,因为交换是在同一位置上进行的。当能够判断出是在同一位置上进行交换时,可省去交换操作,只有在不同位置上交换才是必须的。

进行第三趟时从后面 3 个元素中顺序查找出具有最小排序码的元素  $a[4]$ ,把它的值与  $a[2]$  元素的值交换后,数组  $a$  的当前内容为:

0	1	2	3	4
$R_3$ 27	$R_1$ 35	$R_4$ 44	$R_0$ 48	$R_2$ 64

进行第四趟时从后面两个元素中顺序查找出具有最小排序码的元素  $a[3]$ ,由于待交换的元素也是  $a[3]$ ,所以可省去交换操作,数组  $a$  中记录的排列次序不变。至此整个排序过程结束,数组  $a$  中的记录确实是按照排序码的升序排列了。

下面程序是对具有 Student 类型的结构数组  $a$  中的  $n$  个记录进行选择排序,并输出排序前后的结果。假定排序域是 Student 类型的 num 成员域,它是一个字符串型,两记录排序码之间的比较必须使用字符串函数,不能够直接使用等于号(==);因为这种比较的只是字符指针(即字符数组名)的值,而不是它们指向的字符串。具体程序如下:

```
#include <iomanip.h>
#include <string.h>
struct Student { // 定义学生记录结构
    char num[8]; // 学号
    char name[10]; // 姓名
    short grade; // 成绩
};
Student a[5] = {"cs102", "张平", 78}, {"ch231", "王广敏", 69},
               {"ec115", "刘文", 82}, {"pt327", "古明", 72},
               {"bx214", "张文远", 65}; // 定义全局结构数组 a 并初始化
void output(int n)
// 显示出具有 Student 类型的全局结构数组 a 中的 n 个记录
{
    cout << "显示具有 Student 结构的" << n << "个记录:" << endl;
```

```

        cout.setf(ios::left); // 使向屏幕输出的数据按左对齐显示,
        // 默认情况是按右对齐显示
        for(int i=0; i<n; i++) {
            cout << setw(8) << a[i].num << setw(12) << a[i].name;
            cout << setw(5) << a[i].grade << endl;
        }
        cout << endl;
    }

void range(int n)
    // 对具有 Student 类型的全局结构数组 a 中的 n 个记录进行选择排序
{
    int k; // 用 k 指向每趟中当前具有最小排序码的元素
    for(int i=1; i<=n-1; i++) // 进行 n-1 趟查找和交换
    {
        k=i-1; // 每趟都要给 k 赋初值为 i-1
        for(int j=i; j<n; j++)
            // 进行一趟顺序比较后,得到具有最小排序码的元素为 a[k]
        {
            if(strcmp(a[j].num,a[k].num)<0) k=j;
        }
        if(k!=i-1) { // 当条件成立时交换 a[i-1] 与 a[k] 的值
            Student x=a[i-1];
            a[i-1]=a[k]; a[k]=x;
        }
    }
}

void main()
{
    output(5);
    range(5);
    output(5);
}

```

该程序的运行结果如下:

显示具有 Student 结构的 5 个记录:

```

cs102  张平      78
ch231  王广敏    69
ec115  刘文      82
pt327  古明      72
bx214  张文远    65

```

显示具有 Student 结构的 5 个记录:

```

bx214  张文远    65
ch231  王广敏    69
cs102  张平      78
ec115  刘文      82
pt327  古明      72

```

例 4. 在不改变结构数组中记录排列次序的情况下,显示按排序码的升序排列的记录。

分析:在例3介绍的选择排序方法中,排序前后数组a中记录的排列次序发生了变化,这是因为在排序过程中需要移动记录(即元素值)。是否有办法既能够使记录按排序码的升序输出,又不改变数组a中记录的位置呢?回答是肯定的,只要另外设置一个具有n个元素的整型数组(假定为b),b数组中每个元素的初值为a数组中对应记录的下标位置(即 $b[i] = i$ ),然后采用选择排序的方法调整数组b中每个元素值的排列次序,使得以 $b[0]$ 为下标位置的记录具有最小的排序码,以 $b[1]$ 为下标位置的记录的排序码大于等于 $a[b[0]]$ 记录的排序码,而小于等于以后面元素 $b[2] \sim b[n-1]$ 分别为下标位置所对应记录的排序码,……,以 $b[n-1]$ 为下标的记录 $a[b[n-1]]$ 具有最大的排序码。这样以在数组b中保存的记录位置的移动代替了记录在数组a中的直接移动,从而不需移动记录也同样达到了排序的目的。

接着例3中分析的例子,数组a中的内容和数组b中的初始状态如下:

	0	1	2	3	4					
a	$R_0$	48	$R_1$	35	$R_2$	64	$R_3$	27	$R_4$	44
b	0	1	2	3	4					
	0	1	2	3	4					

若要对数组a中的记录按整数域值的升序显示输出,进行第一趟时从 $b[0] \sim b[4]$ 中顺序查找出以 $b[3]$ 值为下标位置的记录具有最小排序码27,把 $b[3]$ 的值与 $b[0]$ 的值交换后,得到的数组b为:

	0		1		2		3		4
b	3		1		2		0		4

进行第二趟时从 $b[1] \sim b[4]$ 中顺序查找出以 $b[1]$ 值为下标位置的记录具有最小排序码35,此趟无须交换,数组b保持不变。

进行第三趟时从 $b[2] \sim b[4]$ 中顺序查找出以 $b[4]$ 值为下标位置的记录具有最小排序码44,把 $b[4]$ 的值与 $b[2]$ 的值交换后,得到的数组b为:

	0		1		2		3		4
b	3		1		4		0		2

进行第四趟时从 $b[3] \sim b[4]$ 中顺序查找出以 $b[3]$ 值为下标位置的记录具有最小排序码48,此趟也无须交换,数组b保持上一趟的值不变。至此整个排序结束。按照b数组中下标的次序,输出以 $b[i]$ 值作为数组a中下标位置所对应的记录,则输出结果是按照排序码升序排列的,而数组a中的记录始终保持原有次序不变。

假定需要按Student结构类型的数组a中grade域值的升序显示记录,并且要求不改变数组a中记录的位置,则只要对例3中选择排序算法略加修改就可以得到符合此要求的选择排序算法,具体程序如下:

```
#include <iomanip.h>
```

```

#include < string.h>
struct Student { // 定义学生记录结构
    char num[8]; // 学号
    char name[10]; // 姓名
    short grade; // 成绩
};
Student a[5] = {{ "cs102", "张平", 78 }, { "ch231", "王广敏", 69 },
                { "ec115", "刘文", 82 }, { "pt327", "古明", 72 },
                { "bx214", "张文远", 65 } };
void output(int n)
    // 显示出全局结构数组 a 中的 n 个记录
{
    cout.setf(ios::left);
    for(int i=0; i<n; i++) {
        cout << setw(8) << a[i].num << setw(12) << a[i].name;
        cout << setw(5) << a[i].grade << endl;
    }
    cout << endl;
}

void output1(int *b, int n)
    // 以 b 数组中元素值为下标, 显示出全局数组 a 中的对应记录
{
    cout.setf(ios::left);
    for(int i=0; i<n; i++) {
        cout << setw(8) << a[b[i]].num << setw(12) << a[b[i]].name;
        cout << setw(5) << a[b[i]].grade << endl;
    }
    cout << endl;
}

void rangel(int n)
    // 对具有 Student 类型的全局结构数组 a 中的 n 个记录按 grade
    // 域值的升序显示, 并且不允许改变原有记录的位置
{
    int * b = new int[n]; // 动态分配一个具有 n 个整型元素的数组 b
    for(int i=0; i<n; i++)
        b[i] = i; // 为数组 b 中的每个元素赋初值
    int k; // k 为数组 b 中的下标, b[k] 为每趟中当前具有
           // 最小排序码的元素的位置
    for(i=1; i<=n-1; i++) // 进行 n-1 趟查找和交换
    {
        k = i-1; // 每趟都要给 k 赋初值为 i-1
        for(int j=i; j<n; j++)
            // 进行一趟顺序比较后, 得到具有最小
            // 排序码的元素的下标为 b[k]
        {
            if(a[b[j]].grade < a[b[k]].grade)
                k = j;
        }
        if(k != i-1) { // 当条件成立时交换 b[i-1] 与 b[k] 的值

```

```

        int x = b[i - 1];
        b[i - 1] = b[k]; b[k] = x;
    }
}
output1(b, n); // 利用位置数组 b 输出数组 a 中的记录
}

void main()
{
    int n = 5;
    cout << "输出数组 a 中的记录" << endl;
    output(n);
    cout << "按记录的 grade 域值的升序输出数组 a 中的记录" << endl;
    range1(n);
    cout << "再一次输出数组 a 中的记录" << endl;
    output(n);
}

```

从下面程序运行结果可以看出,调用 range(n)选择排序算法前后,数组 a 中记录的排列次序保持不变。

输出数组 a 中的记录

cs102	张平	78
ch231	王广敏	69
ec115	刘文	82
pt327	古明	72
bx214	张文远	65

按记录的 grade 域值的升序输出数组 a 中的记录

bx214	张文远	65
ch231	王广敏	69
pt327	古明	72
cs102	张平	78
ec115	刘文	82

再一次输出数组 a 中的记录

cs102	张平	78
ch231	王广敏	69
ec115	刘文	82
pt327	古明	72
bx214	张文远	65

**例 5.** 用一个指针数组保存结构数组中每个元素的地址,通过调整指针数组中每个元素值的次序,使得按指针数组中下标次序访问每个元素值(即指针)所指向结构数组中的元素时,其元素值(即记录)是按某个域值的升序排列的。

分析:假定结构数组用 a 表示,结构指针数组用 b 表示,b 中每个元素 b[i]的初值为 &a[i]。若数组 a 仍为 Student 类型,排序域仍为 grade,则通过指针 b[i]访问所指结构的 grade 域时,则使用间接成员运算符,此访问操作可表示为 b[i] -> grade。编写此题的程序仍然需



要采用选择排序方法,比较两个排序码的关系表达式为  $b[j] \rightarrow \text{grade} < b[k] \rightarrow \text{grade}$ ,而不是例3中的  $a[j].\text{grade} < a[k].\text{grade}$  或例4中的  $a[b[j]].\text{grade} < a[b[k]].\text{grade}$ ;当一趟比较结束后,需要  $b[k]$  与  $b[i-1]$  交换其值,这一点与例4相同。此题程序可在例4程序的基础上稍加修改即可得到,具体程序如下:

```
#include <iomanip.h>
#include <string.h>
struct Student { // 定义学生记录结构
    // 定义结构体
};
Student a[5] = {("cs102", "张平", 78), ("ch231", "王广敏", 69),
                ("ec115", "刘文", 82), ("pt327", "古明", 72),
                ("bx214", "张文远", 65)};
void output(int n)
    // 显示出全局结构数组 a 中的 n 个记录
{
    // 函数体与上一程序相同
}
void output2(Student * b[], int n)
    // 以数组 b 中每个元素值为指针,显示出所指向的记录,
    // 第一个参数也可写为 Student ** b
{
    cout.setf(ios::left);
    for(int i=0; i<n; i++) {
        cout << setw(8) << b[i] -> num << setw(12) << b[i] -> name;
        cout << setw(5) << b[i] -> grade << endl;
    }
    cout << endl;
}
void range2(int n)
    // 对具有 Student 类型的全局结构数组 a 中的 n 个记录,
    // 按 grade 域值的升序输出,并且不改变原有记录位置,
    // 而是通过改变元素地址的排列次序得到
{
    Student ** b = new Student * [n];
    // 动态分配一个具有 n 个元素的结构指针数组 b
    for(int i=0; i<n; i++)
        b[i] = &a[i]; // 为数组 b 中的每个元素赋初值
    int k; // k 为数组 b 中的下标, b[k] 在每趟中指向当前
           // 具有最小排序码的元素
    for(i=1; i<=n-1; i++) // 进行 n-1 趟查找和交换
    {
        k=i-1; // 每趟都要给 k 赋初值为 i-1
        for(int j=i; j<n; j++)
            // 进行一趟顺序比较后,得到具有最小
            // 排序码的元素的地址为 b[k]
        {
            if(b[j] -> grade < b[k] -> grade) k=j;
        }
    }
}
```

```

        if(k != i - 1) { // 当条件成立时交换 b[i - 1]与 b[k]的值
            Student * x = b[i - 1];
            b[i - 1] = b[k]; b[k] = x;
        }
    }
    output2(b, n);
}

void main()
{
    int n = 5;
    cout << "输出数组 a 中的记录" << endl;
    output(n);
    cout << "按记录的 grade 域值的升序输出数组 a 中的记录" << endl;
    range2(n);
    cout << "再一次输出数组 a 中的记录" << endl;
    output(n);
}

```

此程序的运行结果与上一程序相同。

## 7.5 结构与函数

结构是一种类型,它能够使用在允许简单类型使用的所有地方,当然也允许作为函数的参数类型和返回值类型,下面通过例子说明使用情况。

例 1. 从 Student 结构数组中查找某一给定学号的记录,若能够找到,则表明查找成功,返回记录位置(即元素下标),否则返回 -1,表明查找失败。

分析:按照题目要求,可以编写一个函数来实现,假定函数名用标识符 search 表示,函数返回值类型应为整型。函数参数应包括三个:其一为结构数组参数,假定用 s 表示;其二为数组长度(即数组中所含元素的个数)参数,假定用 n 表示;其三为保存给定学号的结构变量参数,假定用 x 表示。查找过程为:从数组 s 中第一个元素 s[0]起,依次使每一个元素的 num 域的值同 x 的 num 域的值(即给定的学号)进行比较,若相等则表明查找成功,返回该元素的下标,否则继续向后比较,直到比较完最后一个元素仍不成功时返回 -1 即可。函数具体定义如下:

```

int search(Student s[], int n, Student x)
// 第一个参数也可用 Student *s 代替
{
    for(int i = 0; i < n; i++)
        if(strcmp(s[i].num, x.num) == 0)
            return i;
    return -1;
}

```

在这个函数中,s 是结构数组参数,它实际上是定义了一个结构指针参数,n 为整型参数,x 为结构变量参数,它们都是值参,均属于本函数的局部变量。在下面的主函数中,采用

search(a,5,x)调用了上述检索函数,当调用执行时,首先把 a 的值(它是数组 a 的首地址,类型为 Student \*)赋给形参 s,把常数 5 赋给形参 n,把 x 的值赋给形参 x,当然在传送实参值之前系统自动为值参变量分配好对应的存储空间;参数传送后接着执行 search 函数体,由于 s[1].num 的值等于 x.num 的值,所以返回 i 的值 1。调用 search 函数后返回到主函数,把返回值赋给变量 k,然后显示出 a[k]元素的值,它就是要检索的记录。

```
void main()
{
    Student a[5] = ({ "cs102", "张平", 78 }, { "ch231", "王广敏", 69 },
                    { "ec115", "刘文", 82 }, { "pt327", "古明", 72 },
                    { "bx214", "张文远", 65 });
    Student x = { "ch231" };
    int k = search(a, 5, x);
    if (k >= 0)
        cout << a[k].num << " " << a[k].name << " " << a[k].grade << endl;
    else
        cout << "学号为" << x.num << "的记录不存在!" << endl;
}
```

主程序运行结果为:

ch231 王广敏 69

例 2. 从 Student 结构数组中更新某一给定学号的记录,若更新成功则返回 1,否则将新记录插入到数组末尾,并修改数组长度为已有长度加 1,同时返回 0 表示完成插入。

分析:此题同样可以用一个函数来实现,假定函数名用 update 表示,函数类型可定义为整型或布尔型。函数参数有三个,分别为结构指针、数组长度和存储更新值的结构变量,假定依次用 s, n 和 x 表示,其中第一个参数 s 应定义为值参,第二个参数 n 应定义为一个引用类型,因为它要带回修改后的数组长度,即反映到实参变量中,最后一个参数可以定义为值参,也可以定义为引用类型。若一个形参为引用类型,则访问的是实参的存储空间,而不需要像值参那样为其分配存储空间和传送实参值。另外,对于引用参数,若只需要在函数中取用其值,而不需要改变它的值,则最好在参数说明前加上关键字 const,这样当进行非法修改时,能够被编译器发现,避免人为的错误。此题的更新过程为:从数组 s 的第一个元素起顺序查找,若查找到 s[i].num 的值与 x.num 的值相等,就用 x 的值更新(即修改)s[i]元素的值并返回 1,否则表明没有查找到待更新的元素,应将 x 的值插入到数组 s 中下标为 n 的位置上,并将 n 的值增 1(即数组长度增 1)后返回 0。函数具体定义如下:

```
int update(Student s[], int& n, const Student& x)
{
    for(int i=0; i<n; i++)
        if(strcmp(s[i].num, x.num) == 0) {
            s[i] = x; // 用 x 值更新 s[i] 的值
            return 1;
        }
    s[n++] = x; // 将 x 值插入到 s 数组中最后一个记录的后面
    return 0;
}
```

在下面程序的主函数中,两次调用了更新函数,第一次调用时形参 *n* 和 *x* 分别成为了实参 *n* 和 *x* 的别名,并且 *x* 不能被修改;第二次调用时形参 *n* 和 *x* 分别成为了实参 *n* 和 *y* 的别名,并且 *x*(对应实参 *y*)不能被修改。另外第一次调用的结果是修改了结构数组 *a* 中的一条记录,第二次调用的结果是向数组 *a* 中添加了一条记录。

```
#include <iomanip.h>
#include <string.h>
struct Student { // 定义学生记录结构
    char num[8]; // 学号
    char name[10]; // 姓名
    short grade; // 成绩
};
void output(Student * s, int n)
// 显示出结构数组 s 中的 n 个记录
{
    cout.setf(ios::left);
    for(int i=0; i<n; i++) {
        cout << i << ' ' << setw(8) << s[i].num << setw(12) << s[i].name;
        cout << setw(5) << s[i].grade << endl;
    }
    cout << endl;
}

int update(Student s[], int& n, const Student& x)
// 更新 s 数组中学号为 x.num 的记录,若不存在
// 则把 x 记录添加到 s 数组中
{
    for(int i=0; i<n; i++)
        if(strcmp(s[i].num, x.num) == 0) {
            s[i] = x; // 用 x 值更新 s[i] 的值
            return 1;
        }
    s[n++] = x; // 将 x 值插入到 s 数组中最后一个记录的后面
    return 0;
}

void main()
{
    Student a[8] = {{"cs102", "张平", 78}, {"ch231", "王广敏", 69},
                    {"ec115", "刘文", 82}, {"pt327", "古明", 72},
                    {"bx214", "张文远", 65}};
    // 为了给插入记录留有空间,应将定义的
    // 数组长度大于初始化元素常量的个数
    Student x = {"pt327", "古明", 86}, y = {"sr203", "田飞", 74};
    // 定义并初始化两个结构变量 x 和 y
    int n=5; // 定义 n 为数组 a 中当前保存的记录个数,
    // 即被称为 a 的当前长度
    if(update(a, n, x) == 1)
        cout << "完成更新操作!" << endl;
    else
```

```

        cout << "完成插入操作!" << endl;
    if(update[a,n,y] == 1)
        cout << "完成更新操作!" << endl;
    else
        cout << "完成插入操作!" << endl;
    output(a,n); // 输出数组 a 中当前保存的全部记录
}

```

该程序运行结果如下:

```

完成更新操作!
完成插入操作!
0 cs102  张平      78
1 ch231  王广敏    69
2 ec115  刘文      82
3 pt327  古明      86
4 bx214  张文远    65
5 sr203  田飞      74

```

从运行结果可以看出,学号为 pt327 的记录被更新为新值,最后一条记录是新插入的。

例 3. 假定从 Student 结构数组中查找学号等于给定值的记录时,若查找成功则返回该元素的指针,否则返回空。

此题对应的函数如下:

```

Student * search(Student s[], int n, const Student& x)
{
    for(int i=0; i<n; i++)
        if(strcmp(s[i].num, x.num) == 0)
            return &s[i]; // 返回元素的地址
    return 0; // 返回空值,或使用符号常量 NULL
}

```

此函数返回的是结构指针,即被检索到的元素的地址。当一个函数的返回类型为指针类型或引用类型时,则返回语句中使用的变量不能是本函数中定义的局部变量,因为当退出该函数后,其局部变量就不存在了。

假定用下面的主函数调用上述函数。

```

void main()
{
    Student a[8] = {{"cs102","张平",78}, {"ch231","王广敏",69},
                    {"ec115","刘文",82}, {"pt327","古明",72},
                    {"bx214","张文远",65}};

    int n=5;
    Student x, *p;
    cout << "请输入一个待查学生的学号:";
    cin >> x.num;
    p = search(a,n,x);
    if(p!=NULL) {
        cout << p->num << " " << p->name << " " << p->grade << endl << endl;
        cout << "请输入学号为" << x.num << "学生的新成绩:";
        cin >> p->grade;
    }
}

```

```

    }
    else
        cout << "没有找到学号为" << x.num << "的记录" << endl;
    output(a,n);
}

```

假定要找的学号为 bx214,要把他的成绩修改为 86,则程序输入和运行结果如下:

请输入一个待查学生的学号:bx214

bx214 张文远 65

请输入学号为 bx214 学生的新成绩:86

```

0 cs102  张平      78
1 ch231  王广敏    69
2 ec115  刘文      82
3 pt327  古明      72
4 bx214  张文远    86

```

当一个函数返回的是指针或引用时,用户既可以直接从返回指针所指向的对象中或从返回引用所对应的对象中取值,也可以改变它们的值,也就是说,它们既可以当左值也可以当右值使用,否则只能把返回值作为右值使用。

## 7.6 结构与链表

在结构类型中有一种特殊类型,它除了包含有一般的数据域以外,还包含有一个指向自身结构的指针域。这种类型的对象又称为结点,每个结点的指针域用来指向下一个结点,由此形成一个链表。如假定 IntNode 结构(结点)类型为:

```

struct IntNode {
    int data; // 结点值域
    IntNode * next; // 结点指针域
};

```

该类型结点的值域 data 用于存储一个整数,指针域 next 用于存储下一个结点的地址,或者说用于指向下一个结点,当一个结点不需要指向任何结点时,则它的指针域应被置为空。通过 next 指针域使每个结点依次链接起来形成链表。假定具有 IntNode 类型的四个结点,其值分别为 48、56、72 和 83,并且它们依次被链接起来,则对应的链表结构示意图如图 7-1 所示。



图 7-1 一个链表结构的示意图

在一个链表中,指向第一个结点的指针称为表头指针,第一个结点又称为表头结点,每个结点的指针域所指向的结点称为该结点的后继结点,而该结点又称为后继结点的前驱结点,链表中的第一个结点无前驱结点,而最后一个结点(又称为表尾结点)无后继结点。

在图 7-1 中,  $f$  为表头指针,  $f$  所指向的值为 48 的结点为表头结点, 它的后继结点是值为 56 的结点, 此结点的后继结点是值为 72 的结点, 而值为 72 结点的后继是值为 83 的结点, 它是链表中的最后一个结点, 即表尾结点。

当访问一个链表时, 必须从表头指针出发顺序进行, 只有第一个结点被访问后, 才能根据第一个结点的指针域的值访问第二个结点, 同样只有第二个结点被访问后, 才能够访问第三个结点, 依次类推, 表尾结点只能最后被访问到。因此链表具有顺序存取特性, 不像数组那样具有随机存取特性, 即在数组中能够根据下标存取任一元素。

链表能够用来存储同一类型的一组数据, 每个数据保存在一个结点的值域中, 通过结点的指针域建立起数据之间的线性关系。当一组数据需要经常变化时, 即需要不断进行插入或删除时, 则适合采用链表结构, 因为当向链表插入或删除一个结点非常方便, 只需要修改相关指针即可。

链表中的结点通常是通过使用 `new` 运算符动态分配产生的, 若不通过使用 `delete` 运算符及时回收结点, 则动态分配的结点直到程序运行结束才会被系统回收掉, 并且在整个程序运行期间都可以随时访问动态结点中的内容。

下面的 `create` 函数能够根据从键盘上输入的  $n$  个整数建立一个具有  $n$  个结点、每个结点为 `IntNode` 类型的链表。

```
void create(IntNode* & f, int n)
// 建立以 f 为表头指针的具有 n 个结点的链表, f 必须为引用参数,
// 这样才能够使对应的实参成为该链表的表头指针, 以便在返回
// 后的调用函数中使用。
{
    if(n < 0) {
        cerr << "n 的值无效!" << endl;
        exit(1); // 退出程序运行, 此函数在 stdlib.h 头文件中声明
    }
    if(n == 0) { f = NULL; return; } // 置表头指针为空后返回
    cout << "从键盘上输入" << n << "个整数:" << endl;
    int x;
    cin >> x; // 从键盘上输入第一个整数到 x
    f = new IntNode; // 产生一个动态结点作为表头结点
    f->data = x; f->next = NULL; // 建立表头结点
    if(n == 1) return; // 若条件成立则链表已经建好, 应返回
    IntNode* p = f; // 给 p 赋初值, 使之指向刚建立的表头结点
    for(int i = 1; i <= n - 1; i++)
    { // 循环 n - 1 次, 每次建立一个结点并链接到表尾
        cin >> x; // 从键盘缓冲区中顺序读入一个整数到 x
        p->next = new IntNode; // 向表尾添加一个结点
        p = p->next; // 使 p 指向新添加的表尾结点
        p->data = x; // 把 x 的值赋给表尾结点
    }
    p->next = NULL; // 把整个链表的表尾结点的指针域置空
}
```

在上述算法所建立的链表中, 其结点的次序与键盘上输入数值的次序相同, 因为每次都是向表尾插入结点的。在下面的算法中, 由于每次是向表头插入一个新结点, 所以链表中结

点的次序正好与键盘上输入的整数的次序相反。

```
void create1(IntNode* & f, int n)
// 建立以 f 为表头指针的具有 n 个结点的链表, 并按照从键盘上输入
// 入数值的相反次序链接结点
{
    if(n < 0) {
        cerr << "n 的值无效!" << endl;
        exit(1);
    }
    if(n == 0) {f = NULL; return;} // 置表头指针为空后返回
    cout << "从键盘上输入" << n << "个整数:" << endl;
    int x;
    cin >> x; // 从键盘上输入第一个整数到 x
    f = new IntNode; // 产生一个动态结点作为表头结点
    f->data = x; f->next = NULL; // 建立表头结点
    if(n == 1) return; // 若条件成立则链表已经建好, 应返回
    IntNode* p;
    for(int i = 1; i <= n - 1; i++)
    { // 循环 n - 1 次, 每次建立一个结点并链接到表头
        cin >> x;
        p = new IntNode; // 由 p 指针指向一个新分配的结点
        p->data = x; // 把 x 的值赋给 p 结点的值域
        p->next = f; // 把 f 所指向的链表链接到 p 结点的指针域,
        // 相当于把 p 结点插入到 f 链表的表头
        f = p; // 使 f 表头指针指向刚插入的 p 结点
    }
}
```

为了实现同一个目标可以编写出不同的算法, 如将上述算法中从 int x 语句开始向下的程序段改写成下面程序段, 不但正确而且更简练。

```
f = NULL;
while(n -- > 0) {
    IntNode* p = new IntNode; // 由 p 指针指向一个新分配的结点
    cin >> p->data;
    p->next = f;
    f = p; // 使 f 表头指针指向刚插入的 p 结点
}
```

下面是一个遍历链表的算法, 即从表头指针开始, 顺着每一个结点的链(即指针域的值)访问每一个结点。

```
void traverse(IntNode* f)
// 遍历由表头指针 f 所指向的链表
{
    while(f) {
        cout << f->data << ' '; // 输出结点值
        f = f->next; // 得到指向下一个结点的指针
    }
    cout << endl;
}
```



下面的主函数将调用上面的三个函数。

```
void main()
{
    IntNode * head1 = NULL, * head2 = 0;
    int n;
    cout << "输入结点数:";
    cin >> n;
    create(head1, n);
    traverse(head1);
    create1(head2, n);
    traverse(head2);
}
```

程序输入和运行结果如下:

```
输入结点数:6
从键盘上输入 6 个整数:
2 4 6 8 10 12
2 4 6 8 10 12
从键盘上输入 6 个整数:
2 4 6 8 10 12
12 10 8 6 4 2
```

## 7.7 结构与操作符重载

在C++语言中,系统为各种基本数据类型定义了各种相应的操作符,利用它们能够对相应类型上的数据进行运算。如在整数类型上定义了+、-、\*、/、%等算术操作符,==、!=、>=、>、<、<=等关系操作符,=、+=、-=、\*=、/=、%=等赋值操作符,>>和<<用于输入输出流的操作符,以及其他操作符。对于自定义类型,系统只定义了访问成员操作符和赋值操作符,没有定义其他任何操作符,若用户需要对自定义类型的对象进行某一种算术、关系或输入输出等操作,则一种方法是按照一般函数定义的规则定义相应的函数,另一种方法是按照操作符重载函数的定义规则定义相应的操作符重载函数。在C++中除了个别操作符(如直接访问成员操作符.和类作用域指明符::)外,每个操作符都能够被重载,按照需要在自定义类型上定义出相应的操作。但操作符被重载后,其操作符的优先级和结合性仍为系统规定的那样,不会因此而改变。

例1. 假定把一个分数定义为一种结构类型,类型名用Fraction表示,它包含有分子和分母两个整数成员,成员名分别用nume和deno表示,则具体定义格式为:

```
struct Fraction { // 定义分数类型
    int nume; // 定义分子
    int deno; // 定义分母
};
```

下面首先按照一般函数的定义方式给出在分数类型上的一些典型操作,然后再按照操作符重载函数的定义方式给出相应的操作。

为了用一个函数实现两个分数的加法,假定函数名用 `FranAdd` 表示,该函数应带有两个分数类型的参数,分别表示被加数和加数,函数的返回值也应为分数类型,以便返回相加的结果分数。该函数具体定义如下:

```
Fractioin FranAdd(const Fractioin& a, const Fractioin& b)
// 返回两个分数 a 和 b 之和
{
    Fractioin c; // 定义临时变量 c,用于保存求和结果
    c.nume = a.nume * b.deno + b.nume * a.deno; // 计算结果分数的分子
    c.deno = a.deno * b.deno; // 计算结果分数的分母
    FranSimp(c); // 对结果分数进行简化处理
    return c; // 返回结果分数
}
```

在上述前 3 条语句计算出结果分数 `c` 之后,还需要调用 `FranSimp` 函数对其进行简化,使得 `c` 变为最简分数,并且当分母为负时使其变为正,把符号加到分子上。函数具体定义如下:

```
void FranSimp(Fractioin& x)
// 把 x 化简为最简分数
{
    // 求 x 分数的分子和分母的最大公约数
    int m, n, r;
    m = x.nume; n = x.deno;
    // 把 x 分数的分子和分母分别赋给 m 和 n 作为其初值
    r = m % n; // 将 m 整除以 n 的余数赋给 r 作为其初值
    while(r != 0)
    { // 当循环结束后, n 的值就是 x 的分子和分母的最大公约数
        m = n; n = r;
        r = m % n;
    }
    // 化简 x,使分子和分母均缩小 n 倍
    if(n != 1) {
        x.nume /= n;
        x.deno /= n;
    }
    // 若分母为负则让分子和分母同时取负后使分母转换为正值
    if(x.deno < 0) {
        x.nume = -x.nume;
        x.deno = -x.deno;
    }
}
```

在上面函数中,求两个整数 `m` 和 `n` 的最大公约数采用的是辗转相除取余法,直到 `r` 为 0 时,`n` 的当前值就是它们的最大公约数。例如,要求 32 和 12 的最大公约数,第一次用 32 整除以 12 得余数为 8,第二次用 12 整除以 8 得余数为 4,第三次用 8 整除以 4 得余数为 0,此时的 `n` 值 4 就是 32 和 12 的最大公约数。

为了用一个函数判断两个分数是否相等,假定函数名用 `FranEqual` 表示,该函数应返回一个布尔值或整型值,若两个分数相等则返回 1,否则返回 0。函数具体定义如下:

```

bool FranEqual(const Frac &a, const Frac &b)
// 若 a 和 b 的值相等则返回 true, 否则返回 false
{
    if(a.nume * b.deno - b.nume * a.deno == 0) return true;
    else return false;
}

```

当进行一个分数的输入和输出时,也可以定义为函数的形式,假定分数输入或输出的格式为:分子/分母。如要输入一个分数 $\frac{8}{15}$ ,则采用 8/15 的格式输入;若一个分数 x 的分子和分母分别为 -4 和 5,则输出结果为: -4/5。假定进行分数输入和输出的函数名分别用 FranInput 和 FranOutput 表示,则它们的具体定义分别如下:

```

void FranInput(Frac &x)
// 从键盘上按规定格式输入一个分数到 x 中
{
    char ch; // 用 ch 保存分数输入中的除号
    cout << "Input a fraction:";
    cin >> x.nume >> ch >> x.deno;
    if(x.deno == 0) {
        cerr << "除数为 0!" << endl;
        exit(1); // 中止程序运行,返回 C++ 主操作窗口
    }
}

```

```

void FranOutput(Frac &x)
// 按规定格式输出 x 中的分数
{
    cout << x.nume << "/" << x.deno << endl;
}

```

假定使用如下主函数调用上述各函数。

```

void main()
{
    Frac a, b, c; // 定义 a, b, c 三个分数对象
    FranInput(a); // 输入分数 a
    FranInput(b); // 输入分数 b
    c = FranAdd(a, b); // a 和 b 相加结果赋给 c
    cout << "a: "; FranOutput(a); // 输出分数 a
    cout << "b: "; FranOutput(b); // 输出分数 b
    cout << "c: "; FranOutput(c); // 输出分数 c
    if(FranEqual(a, b)) // 判断 a 和 b 是否相等
        cout << "a == b" << endl;
    else
        cout << "a != b" << endl;
    if(FranEqual(c, FranAdd(a, b)))
        // 判断 c 和 a 加 b 的结果对象是否相等
        cout << "c == a + b" << endl;
    else
        cout << "c != a + b" << endl;
}

```

}

假定分别向分数 a 和 b 输入的具体值为 8/15 和 4/5, 则程序运行结果如下:

```
Input a fracion:8/15
Input a fracion:4/5
a: 8/15
b: 4/5
c: 4/3
a!=b
c==a+b
```

利用操作符(运算符)重载函数也可以实现上述每个函数的功能, 并且调用格式非常方便, 与在标准类型的对象上使用操作符的格式完全相同。操作符重载函数的函数名为关键字 operator 及后跟一个操作符, 该操作符就是重载操作符, 它同前面的关键字 operator 之间有无空格均可。操作符重载函数必须带有参数, 并且至少要有一个为用户自定义类型的参数。对于单目操作符的重载, 其参数表中为一个参数, 该参数就是单目操作符的运算对象; 对于双目操作符的重载, 其参数表中有两个参数, 第一个参数为双目操作符左边的运算对象, 第二个参数为双目操作符右边的运算对象。

单目操作符重载函数的定义格式为:

<返回类型> operator <单目操作符> (<一个用户类型参数说明>) {<函数体>}

双目操作符重载函数的定义格式为:

<返回类型> operator <双目操作符> (<第一个参数说明>, <第二个参数说明>)  
{<函数体>}

单目操作符重载函数的调用格式为:

<单目操作符> <实参>

它等价于下面的调用格式:

operator <单目操作符> (<实参>)

特别地, 对于单目后增 1 或减 1 操作符, 在重载函数定义格式的参数表中要增加一个整型参数, 以视同前增 1 或减 1 操作符重载格式相区别, 但该整型参数是虚设的, 可以只给出整型关键字而不给出参数名。

双目操作符重载函数的调用格式为:

<第一个实参> <双目操作符> <第二个实参>

它等价于下面的调用格式:

operator <双目操作符> (<第一个实参>, <第二个实参>)

我们已经介绍过两个分数相加的一般函数格式和调用方式, 下面我们用加号操作符重载函数来定义同样的功能。

```
Fracion operator+ (const Fracion& a, const Fracion& b)
// 返回两个分数 a 和 b 之和
```

```

{
    Franction c; // 定义临时变量 c, 用于保存求和结果
    c.nume = a.nume * b.deno + b.nume * a.deno; // 计算结果分数的分子
    c.deno = a.deno * b.deno; // 计算结果分数的分母
    FranSimp(c); // 对结果分数进行简化处理
    return c; // 返回结果分数
}

```

类似地,对于判断两个分数是否相等的操作,同样可以通过定义等于操作符重载函数来实现,函数具体定义如下:

```

bool operator == (const Franction& a, const Franction& b)
// 若 a 和 b 的值相等则返回 1, 否则返回 0
{
    if(a.nume * b.deno == b.nume * a.deno) return true;
    else return false;
}

```

对于分数的输入操作,也可以通过定义提取操作符(>>)重载函数来实现,该重载函数的第一个参数为标准输入流类型 istream 的引用,第二个参数为分数类型的引用,函数返回类型应为标准输入流类型的引用,以便能够继续使用提取操作符输入其他数据。针对分数结构类型的提取操作符重载函数的定义如下:

```

istream& operator >> (istream& istr, Franction& x)
// 从键盘上按规定格式输入一个分数到 x 中
{
    char ch; // 用 ch 保存分数输入中的除号
    cout << "Input a fracion:";
    istr >> x.nume >> ch >> x.deno;
    if(x.deno == 0) {
        cerr << "除数为 0!" << endl;
        exit(1);
    }
    return istr;
}

```

因为标准输入流对象 cin 是具有 istream 类型的,所以当用 cin >> a >> b 语句(假定 a 和 b 均为一个分数对象)调用上述重载函数时,首先将 cin 传送给第一个参数 istr,把 a 传送给第二个参数 x,因此 istr 和 x 就分别成了 cin 和 a 的别名,该函数返回的是被传送的实参对象 cin,接着执行 cin >> a >> b 语句中的第二个提取操作符时,仍调用上述的重载函数,分别把 cin 和 b 传送给形参 istr 和 x,使它们成为 cin 和 b 的别名,在函数体中给 b 输入数据后,又返回 cin 对象。

对于分数的输出操作,对应的插入操作符(<<)重载函数如下:

```

ostream& operator << (ostream& ostr, Franction& x)
// 按规定格式输出 x 中的分数
{
    ostr << x.nume << '/' << x.deno << endl;
    return ostr;
}

```

```
}
```

因为 `cout` 是标准输出流类型 `ostream` 的对象, 当执行 `cout << a << endl` 语句(假定 `a` 为分数对象)时, 将自动调用上述插入操作符重载函数, 在函数体中按照规定格式输出 `a` 的值后, 返回 `cout` 对象, 以便能够在一条输出语句中多次使用插入操作符输出各种数据。

为了实现一个分数前缀加的操作, 可进行如下单目前缀加操作符重载函数的定义:

```
Frunction& operator ++ (Frunction& x)
// x 先增 1, 然后返回它的引用
{
    x.nume += x.deno;
    return x;
}
```

在这个函数中, 需要修改引用参数 `x` 的值, 所以它不能用 `const` 关键字修饰。该函数返回的是一个分数类型的引用, 由返回语句可知, 返回的是 `x` 的引用, 也就是返回调用该函数的实在变量, 这样当一个分数进行前缀加操作后, 仍可作为左值使用, 从而符合前缀加操作符的定义。对于前缀减及各种赋值操作符的重载也是如此。

为了实现对一个分数后缀加的操作, 可进行如下单目后缀加操作符重载函数的定义:

```
Frunction operator ++ (Frunction& x, int)
// 使 x 增 1, 但返回的是 x 的原值
{
    Frunction y = x; // 保存 x 的值以便返回
    x.nume += x.deno;
    return y;
}
```

下面给出调用上述操作符重载函数的主函数。

```
void main()
{
    Frunction a, b, c; // 定义 a, b, c 三个分数对象
    cin >> a >> b; // 输入分数 a 和 b
    FranSimp(a); FranSimp(b); // 对 a 和 b 进行规范化
    c = a + b; // a 和 b 相加结果赋给 c
    cout << "a: " << a; // 输出分数 a
    cout << "b: " << b; // 输出分数 b
    cout << "c: " << c; // 输出分数 c
    if(a == b) // 判断 a 和 b 是否相等
        cout << "a == b" << endl;
    else
        cout << "a != b" << endl;
    if(c == a + b) // 判断 c 和 a 加 b 的结果对象是否相等
        cout << "c == a + b" << endl;
    else
        cout << "c != a + b" << endl;
    c = a++; cout << "c = " << c; cout << "a = " << a; // 使用后缀加
    c = ++a; cout << "c = " << c; cout << "a = " << a; // 使用前缀加
}
```

在主函数中使用了如下一些操作符表达式调用相应的操作符重载函数

- (1) `cin >> a`
- (2) `cout << a`
- (3) `a + b`
- (4) `a == b`
- (5) `c == a + b`
- (6) `a ++`
- (7) `++ a`

它们分别与下面的调用表达式等效,显然上面的调用格式是简便和自然的。

- (1) `operator >> (cin, a)`
- (2) `operator << (cout, a)`
- (3) `operator + (a, b)`
- (4) `operator == (a, b)`
- (5) `operator == (c, operator + (a, b))`
- (6) `operator ++ (a, 1)` // 第二个实参可以为任意整数值,这里用 1 表示
- (7) `operator ++ (a)`

主函数执行结果如下:

```
Input a fraction:8/15 // 假定向 a 输入的分数值为 8/15
Input a fraction:4/5  // 假定向 b 输入的分数值为 4/5
a: 8/15
b: 4/5
c: 4/3
a != b
c == a + b
c = 8/15
a = 23/15
c = 38/15
a = 38/15
```

例 2. 编写一个程序,能够分别从任何标准类型的数组、字符串数组和以前介绍过的 `Student` 结构类型的数组中查找出最大值,假定结构数组中的元素以 `grade` 域值的大小来决定元素的大小。

分析:此程序中需要包含一个求任意类型数组中元素最大值的函数模板,该函数模板的具体定义如下:

```
template<class Type> // 定义模板类型参数为 Type
Type FindMax(Type a[], int n)
// 求类型为 Type 的具有 n 个元素的数组 a 中的最大值并返回
{
    Type max = a[0]; // 用 max 保存已比较元素中的最大值,初值为 a[0]
    for(int i = 1; i < n; i++)
        if(a[i] > max) max = a[i];
    return max; // 返回数组中的最大值
}
```

对于字符串数组,若要求出其最大字符串,则不能使用上面的函数模板生成针对字符指针(即字符串)的函数,因为函数中进行的 `a[i]` 与 `max` 的比较是字符指针的比较,而不是所指数组的比较。因此必须针对字符串数组专门写出求最大值的特定函数,它是函数模板的一个重载函数,当进行求字符串数组中最大值的调用时,将自动调用该函数,而不会由函数模板生成相应的模板函数并被调用。此特定函数的具体定义如下:

```
char * FindMax(char * a[], int n)
// 从具有 n 个元素的字符串数组 a 中查找出最大值并返回
{
    char * max = a[0];
    for(int i = 1; i < n; i++)
        if(strcmp(a[i], max) > 0) max = a[i];
    return max;
}
```

当利用函数模板生成针对 `Student` 结构类型的实例函数时,需要进行两个结构对象 `a[i]` 与 `max` 的大于比较,为此程序中必须提供对两个结构对象进行大于号比较的操作符重载函数的支持。根据题意,该重载操作符函数的定义如下:

```
bool operator > (const Student& x, const Student& y)
// 用 grade 域值的大小来代表元素的大小
{
    return x.grade > y.grade;
}
```

假定需要从整型数组、字符串数组和 `Student` 数组中查找最大值,则完整程序如下:

```
#include <iostream.h>
#include <string.h>
struct Student { // 定义学生记录结构
    char num[8]; // 学号
    char name[10]; // 姓名
    short grade; // 成绩
};
template <class Type> Type FindMax(Type a[], int n);
char * FindMax(char * a[], int n);
int operator > (Student& x, Student& y);

void main()
{
    int a1[8] = {35, 26, 48, 69, 60, 35, 83, 55};
    char * a2[6] = {"GULIANG", "NINGCHEN", "XUXKAI", "WEIRONG",
        "CUICHM", "WANGPING"};
    Student a3[5] = {{"cs102", "张平", 74}, {"ch231", "王广敏", 89},
        {"ec115", "刘文", 62}, {"pt327", "古明", 75},
        {"bx214", "张文远", 68}};
    int b1 = FindMax(a1, 8);
    // 因 a1 的元素类型为 int, 所以将由函数模板生成类型
    // Type 为 int 的实例函数并被调用执行。
    char * b2 = FindMax(a2, 6);
```



```

        // 因 a2 的元素类型为 char *, 所以将调用针对该类型的
        // 特定函数执行, 只有不存在针对一个具体类型的特
        // 定函数时, 才会由模板自动生成。
Student b3 = FindMax(a3, 5);
        // 因 a3 的元素类型为 Student, 所以将由模板函数生成类型
        // Type 为 Student 的实例函数并被调用执行, 在执行中需要
        // 调用大于号操作符重载函数比较两个结构的大小。
cout << b1 << endl;
        // 输出数组 a1 中的最大值
cout << b2 << endl;
        // 输出数组 a2 中的最大值
cout << b3.num << ' ' << b3.name << ' ' << b3.grade << endl;
        // 输出数组 a3 中的最大值
}

template< class Type> // 定义模板类型参数为 Type
Type FindMax(Type a[], int n)
    // 求类型为 Type 的具有 n 个元素的数组 a 中的最大值并返回
{
    Type max = a[0]; // 用 max 保存已比较元素中的最大值, 初值为 a[0]
    for(int i = 1; i < n; i++)
        if(a[i] > max) max = a[i];
    return max; // 返回数组中的最大值
}

char* FindMax(char* a[], int n)
    // 从具有 n 个元素的字符串数组 a 中查找出最大值并返回
{
    char* max = a[0];
    for(int i = 1; i < n; i++)
        if(strcmp(a[i], max) > 0) max = a[i];
    return max;
}

bool operator > (const Student& x, const Student& y)
    // 用 grade 域值的大小来代表元素的大小
{
    return x.grade > y.grade;
}

```

此程序的运行结果如下:

```

83
XUXKAI
ch231 王广敏 89

```

## 7.8 联合

### 7.8.1 联合的定义和访问

联合(union)是又一种用户定义的数据类型, 它同结构类型的定义一样, 也是由若干个

数据成员所组成,并且也可以带有成员函数。但有一点不同:在任一时刻,结构中的所有成员都是可访问的,而联合中只有一个成员是可访问的,其余所有成员都是不可访问的。这种不同反映到存储空间分配上的差别:每个结构对象包含有全部数据成员的存储空间,它所占存储空间的大小等于所有数据成员所占存储空间大小的总和,而每个联合对象所占存储空间的大小等于所有数据成员所占存储空间中的最大值,在任一时刻只能从对象的首地址开始保存一个数据成员的值。

假定一个结构对象  $x$  和一个联合对象  $y$  具有相同的数据成员,即均包含有一个字符成员、一个整数成员、一个整数指针成员和一个双精度浮点数成员,则对应的结构和联合类型定义分别为:

<pre> struct stype {     char ch;     int gr;     int *pt;     double db; };         </pre>	<pre> union utype {     char ch;     int gr;     int *pt;     double db; };         </pre>
---	--

这里用  $ch$  表示字符成员,  $gr$  表示整数成员,  $pt$  表示整数指针成员,  $db$  表示双精度浮点数成员。结构定义的格式是从关键字 `struct` 开始的,后跟结构类型名和结构体,而联合定义的格式是从关键字 `union` 开始的,后跟联合类型名和联合体。结构类型名和联合类型名都是用户命名的类型标识符,结构体和联合体都是由用花括号括起来的若干个数据成员的定义所组成,当然也可以包含有成员函数的声明,两种类型定义的最后都必须用分号结束。

定义具有 `stype` 类型的结构对象  $x$  和定义具有 `utype` 类型的联合对象  $y$  的语句分别如下:

```

stype x;
utype y;
    
```

$x$  和  $y$  对应的存储空间分配分别如图 7-2(a)和(b)所示:

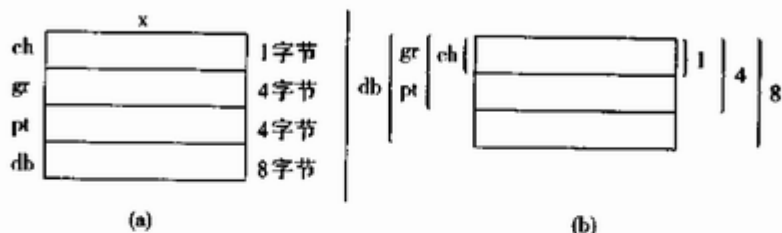


图 7-2  $x$  和  $y$  的存储空间分配示意图

从图 7-2 可以看出:结构对象  $x$  的大小等于所有四个数据成员的大小之和,即等于  $1 + 4 + 4 + 8 = 17$  个字节,而联合对象  $y$  的大小等于所有四个数据成员的大小的最大值,即等于  $\max(1, 4, 4, 8) = 8$  个字节。在作用域范围内,结构对象的每个数据成员都有固定的存储位置,都可以随时被访问,而联合对象的每个数据成员都从同一位置(即对象的首地址)开始存储,在任一时刻只能保存一个数据成员,因而也只有该成员能够被访问,当然在不同时刻可

以用联合对象存储不同的成员并进行相应的访问。联合对象的存储空间的利用率视存储不同的成员而定,例如,当利用  $y$  的  $ch$  成员保存一个字符时, $y$  中只有第一个字节被利用,剩余 7 个字节空闲,当利用  $y$  的  $gr$  或  $pt$  成员保存一个整数或整数指针时, $y$  中只有前 4 个字节被利用,而后 4 个字节空闲,当利用  $y$  的  $db$  成员保存一个双精度浮点数时, $y$  中的 8 个字节全部被利用。

结构与联合在变量的初始化上也有所不同,系统允许对结构中的每个数据成员按照定义的次序进行初始化,但只允许对联合中的第一个数据成员进行初始化,而不允许对其他数据成员进行初始化。当然进行初始化的数据要用花括号括起来。

联合变量的定义格式也同使用结构一样,包括用联合类型名定义、在定义联合类型的同时定义和定义无名联合类型的同时定义这三种情况。

对联合对象中成员的访问也包括使用点运算符进行直接成员访问和使用箭头运算符进行间接成员访问这两种方式。如对于上述的联合对象  $y$ ,直接访问每个成员的表示为  $y.ch$ ,  $y.gr$ ,  $y.pt$  和  $y.db$ ,若  $p$  是指向上述  $utype$  类型的指针类型,则间接访问  $p$  所指对象中每个成员的表示为  $p \rightarrow ch$ ,  $p \rightarrow gr$ ,  $p \rightarrow pt$  和  $p \rightarrow db$ ,若表示成直接访问操作则相应为  $(*p).ch$ ,  $(*p).gr$ ,  $(*p).pt$  和  $(*p).db$ 。

在联合类型的定义中,若既没有给出类型名又没有给出变量也是有意义的,此时联合中的成员可以直接使用。这种联合被称为匿名联合或无名联合,通常被定义在一个结构或类(class)类型的内部,联合中的成员直接作为所在结构或类中的成员使用。例如:

```
struct ABC {
    char ch;
    union { // 匿名联合
        int ia;
        float fa;
    };
    ABC *pa;
} x, *px = &x;
```

由于  $ia$  和  $fa$  被定义为匿名联合中的成员,任一时刻只有一个成员有效,所以在任何时刻  $ABC$  类型中只包含三个成员: $ch$ ,  $ia$  和  $pa$ ,或者  $ch$ ,  $fa$  和  $pa$ 。 $ABC$  类型的大小为 9 个字节,其中一个字节用于  $ch$  成员,接着 4 个字节用于无名联合中的  $ia$  或  $fa$  成员,最后 4 个字节用于  $pa$  成员。在  $ABC$  类型的定义中,同时定义了该类型的变量  $x$  和指针变量  $px$ ,并将  $px$  初始化为  $x$  的地址。对象  $x$  中的成员可表示为  $x.ch$ ,  $x.ia$  (或  $x.fa$ ) 和  $x.pa$ ,对象  $x$  中的成员通过指向  $x$  的指针  $px$  可表示为  $px \rightarrow ch$ ,  $px \rightarrow ia$  (或  $px \rightarrow fa$ ) 和  $px \rightarrow pa$ 。

## 7.8.2 使用联合举例

例 1. 利用一个数组保存一个单位的职工记录,假定每个职工含有编号、姓名、性别、类别和职级这五个数据项,其中类别取整数 1,2,3 之中的值,用 1 表示干部类别,用 2 表示教师类别,用 3 表示工人类别,在干部类别中分为“JVJI”(局级)、“CHUJI”(处级)、“KEJI”(科级)和“KEYUAN”(科员)四个职级,在教师类别中分为“JIAOSHOU”(教授)、“FUJIAOSHOU”(副教授)、“JIANGSHI”(讲师)和“ZHUJIAO”(助教)四个职级,在工人类别中分为 1-8 共八个职级。

编一程序把表 7-1 中的数据输入到数组中,接着把数组中的记录输出到屏幕上,然后统计出每一类别的人数。

表 7-1 职工记录简表

编号	姓名	性别	类别	职级
01001	Liuminzhu	m	1	CHUJI
01002	Zhaogang	m	1	KEJI
01003	Wangmin	f	2	FUJIAOSHOU
02001	Xuzhongyi	m	2	JIANGSHI
02005	Liziyou	m	3	4
03002	Zhuohong	f	2	ZHUJIAO
03014	Chenyi	m	3	5
12020	Dingrong	f	2	JIANGSHI

分析:由题意可知,应把职工记录定义为结构类型,该类型的前四个成员应分别为编号(num)、姓名(name)、性别(sex)和类别(kind),其中编号和姓名均为字符串型,性别定义为字符型,假定用字母 m 表示男,用字母 f 表示女,类别应定义为短整型,该结构的最后一个数据成员应定义为无名联合类型,它包含有三个数据成员,分别为干部(cadre)、教师(teacher)和工人(worker),对应类型分别为字符串、字符串和短整型,根据一个记录中的不同类别将使用联合中的不同成员。职工记录的结构类型定义如下:

```
struct Workers { // 职工记录类型
    char num[6]; // 编号
    char name[12]; // 姓名
    char sex; // 性别
    short kind; // 类别
    union { // 职级
        char cadre[8]; // 干部职级
        char teacher[12]; // 教师职级
        short worker; // 工人职级
    };
};
```

从键盘上向具有 Workers 类型的数组中输入 n 个记录的函数定义如下:

```
void Input(Workers a[], int n)
// 向结构数组 a 中输入 n 个职工记录
{
    for(int i=0; i<n; i++) {
        cout << "请输入第" << i+1 << "条记录:" << endl;
        cin >> a[i].num >> a[i].name >> a[i].sex;
        cin >> a[i].kind;
        switch(a[i].kind) // 根据类别输入相应的职级
        {
            case 1:
                cin >> a[i].cadre;
                break;
```

```

        case 2:
            cin >> a[i].teacher;
            break;
        case 3:
            cin >> a[i].worker;
    }
}
}

```

从具有 Workers 类型的数组中向屏幕上输出 n 个记录(假定不输出类别的值)的函数定义如下:

```

void Output(Workers a[], int n)
// 把数组 a 中的 n 个记录输出到屏幕上
{
    // 使每项数据按左对齐输出
    cout.setf(ios::left);
    // 输出表头,即记录中的各项名称
    cout << setw(6) << "num";
    cout << setw(12) << "name";
    cout << setw(5) << "sex";
    cout << setw(12) << "duty" << endl;
    // 依次输出每条记录
    for(int i=0; i<n; i++) {
        cout << setw(6) << a[i].num;
        cout << setw(12) << a[i].name;
        cout << setw(5) << a[i].sex;
        cout << setw(12);
        switch(a[i].kind) {
            case 1:
                cout << a[i].cadre;
                break;
            case 2:
                cout << a[i].teacher;
                break;
            case 3:
                cout << a[i].worker;
        }
        cout << endl; // 输出每条记录后换行
    }
}

```

从数组 a 中统计出各类别人数的函数定义如下:

```

void Count(Workers a[], int n)
// 统计出数组 a 中各类别的人数并输出
{
    int c1,c2,c3; // 用它们分别统计干部、教师和工人的人数
    c1=c2=c3=0;
    for(int i=0; i<n; i++) {
        switch(a[i].kind) {
            case 1: c1++; break;

```

```

        case 2: c2 ++; break;
        case 3: c3 ++; break;
    }
}
cout << "cadres: " << c1 << endl;
cout << "teachers: " << c2 << endl;
cout << "workers: " << c3 << endl;
}

```

按照题目要求,主函数编写如下:

```

void main()
{
    int n=8; // n 等于表 7-1 中的记录数
    Workers * a=new Workers[n]; // 动态分配数组
    Input(a,n);
    cout << endl;
    Output(a,n);
    cout << endl;
    Count(a,r);
    delete []a;
}

```

该程序运行结果如下(键盘输入除外):

num	name	sex	duty
01001	liuminzhu	m	CHUJI
01002	zhaogang	m	KEJI
01003	wangmin	f	FUJIAOSHOU
02001	xuzhongyi	m	JIANGSHI
02005	liziyou	m	4
03002	zhuhong	f	ZHUJIAO
03014	chenyi	m	5
12020	dingrong	f	JIANGSHI

```

cadres: 2
teachers: 4
workers: 2

```

例 2. 假定一种结点的结构如下:

```

struct MixNode {
    short mark; // 结点值类型标志域
    union {
        float f; // 浮点型值域
        char * r; // 字符串型值域
    };
    MixNode * next; // 链接指针域
};

```

该结点的值域为无名联合中的 f 或 r, 当标志域 mark 分别取 1 或 2 时, 对应的值域分别为 f 或 r, 当然在任一时刻只能有一个值域, 不是 f 就是 r。该结点的指针域为 next, 由它把同

一类型的不同结点链接起来。编一程序,动态产生每个结点,其值由键盘输入,并按照结点产生的先后次序链接起来形成一个链表,然后遍历这个链表。

此题的完整程序如下,请同学们自行分析。

```
#include<iostream.h>
#include<string.h>
#include<stdlib.h>
struct MixNode {    // 结构定义
    short mark;      // 结点值类型标志域
    union {
        float f;    // 浮点型值域
        char * r;    // 字符串型值域
    };
    MixNode * next;  // 链接指针域
};

void Create(MixNode * &, int); // Create 函数声明
void Traverse(MixNode * );    // Traverse 函数声明
void main()
{
    MixNode * f;
    int n=6; // 假定建立具有 6 个结点的链表
    Create(f,n);
    Traverse(f);
}

void Create(MixNode * & head, int n)
// 建立以 head 为表头指针的一个链表,结点值由键盘输入
{
    if(n<0) {
        cerr << "n 的值无效!" << endl;
        exit(1);
    }
    MixNode * p = new MixNode; // 首先为链表建立一个附加表头结点
    head = p; // 将 p 赋给表头指针
    for(int i=0; i<n; i++) {
        p->next = new MixNode; // 将新结点的地址赋给 p 结点的
        // 指针域,接着赋给 p
        cout << "输入结点值类型的标记(1:浮点, 2:字符串):";
        cin >> p->mark;
        if(p->mark == 1) {
            cout << "输入一个浮点数:";
            cin >> p->f;
        }
        else {
            cout << "输入一个字符串:";
            char a[20]; // 用字符数组 a 暂存输入的字符串
            cin >> a;
            p->r = new char[strlen(a) + 1];
            // 为 p 结点的 r 字符指针动态分配字符串空间,
        }
    }
}
```

```

        // 其大小等于输入字符串长度加 1
        strcpy(p -> r, a);
        // 将数组 a 中保存的字符串拷贝到 p 结点的 r 指针
        // 所指向的存储空间中
    }
}
p -> next = NULL; // 将链表中最后一个结点的指针域置空
head = head -> next;
// 将指向链表的第一个结点的指针赋给 head 带回
}

void Traverse(MixNode * head)
// 遍历以 head 为表头指针的链表
{
    MixNode * p = head;
    while(p != NULL) {
        if(p -> mark == 1)
            cout << p -> f << ' ';
        else
            cout << p -> r << ' ';
        p = p -> next;
    }
    cout << endl;
}

```

## 习题七

(一) 计算出下列每个结构类型的大小

1. struct AA {  
    int \*a;  
};
2. struct BB {  
    int a;  
    int b;  
};
3. struct CC {  
    char \*data;  
    BB s;  
    CC \*link;  
};
4. struct DD {  
    short list1[5];  
    AA list2[5];  
    int len;  
};



```

5. struct EE {
    char h;
    union {
        int b;
        double c;
        char d[6];
    };
    CC *a[2];
};

6. struct FF {
    struct {
        int ac, *bc;
    } st, *pt;
    float *fb;
};

```

(二) 假定 a,b,c,d,e,f 依次为(一)中定义的类型对象,请写出每个对象中的所有成员表示,若成员仍为结构、结构指针、数组、结构数组或指针数组,则同时要写出它们每个成员(元素)或所指成员的表示(指向自身结构的指针除外)

(三) 请指出下面每个函数的功能

```

1. float Average(Person a[], int n)
{
    int x=0;
    for(int i=0; i<n; i++)
        x += a[i].age;
    return float(x)/n;
}

2. void Count(Person a[], int n, int &c1, int &c2)
{
    c1 = c2 = 0;
    for(int i=0; i<n; i++)
        if(a[i].sex == true)
            c1++;
        else
            c2++;
}

3. Student FindMix(Student *a, int n)
{
    if(n <= 0) {
        cerr << "a 中没有记录,停止运行!" << endl;
        exit(1);
    }
    int k=0;
    for(int i=1; i<n; i++)
        if(a[i].grade < a[k].grade)
            k = i;
}

```

```

        return a[k];
    }

4. IntNode* FindMax(IntNode * f)
{
    if(f) return NULL;
    IntNode * p = f;
    f = f -> next;
    while(f) {
        if(f -> data > p -> data)
            p = f;
        f = f -> next;
    }
    return p;
}

5. int operator == (Student x, char* key)
{
    if(strcmp(x.num, key) == 0)
        return 1;
    else
        return 0;
}

6. Fracnode operator * (Fracnode& x, Fracnode& y)
{
    Fracnode z;
    z.num = x.num * y.num;
    z.deno = x.deno * y.deno;
    FranSimp(z);
    return z;
}

7. Fracnode operator * (int c, Fracnode& x)
{
    Fracnode y;
    y.num = c * x.num;
    y.deno = x.deno;
    return y;
}

8. int Count(MixNode* f, int& n)
{
    n = 0;
    int m = 0;
    while(f) {
        m++;
        if(f -> mark == 1)
            n++;
        f = f -> next;
    }
    return m;
}

```

#### (四) 编写下列程序或函数

1. 编一函数,从类型为 Person 的、具有  $n$  个人员记录的数组  $a$  中查找并打印出年龄不小于整型变量  $x$  值的所有记录,要求所有输出记录的同一数据项具有相同的显示宽度。
2. 分别编写一个普通函数和减法操作符重载函数,实现具有 Fraction 类型的两个分数相减的操作,运算结果保存在第一个操作数中并带回,同时函数返回第一个操作数的引用。
3. 编写一个函数,实现从任意类型的具有  $n$  个元素的数组中删除下标为  $i$  ( $0 \leq i < n-1$ ) 的元素值并返回这个值的功能。

注意:具体删除操作是把下标为  $i+1 \sim n-1$  的元素值依次向前移动一个位置,并使元素个数变为  $n-1$ 。

4. 编写一个函数,对于具有 Workers 类型和  $n$  个元素的数组  $a$ ,若从下标为  $k$  的位置起向后查找属于教师类别的、职级为  $x$  值的元素,当查找成功时返回该元素的下标,否则返回  $-1$ 。

## 第八章 类与对象

类(class)是对具有共同属性和行为的一类事物的抽象描述,共同属性被描述为类中的数据成员,共同行为被描述为类中的成员函数。虽然类所描述的事物具有共同的属性和行为(操作),但每一个具体事物(又称为个体、实例或对象)都具有属于自己的属性值和行为特征。例如,人的共同属性有姓名、性别、出生日期、民族、籍贯等,但每个人都有自己的姓名、性别等属性值,他与别人的属性值可以不同也可以相同,如允许有重名,性别只有男、女之分等;人的共同行为有工作、学习、休息、吃饭、穿衣、打扮、爱好等,但每个人都有属于自己的行为特征,当然也允许与别人的行为特征相同或不同,如有的人爱好看电视,有的人爱好听音乐,有的人爱好下象棋,有的人爱好游泳等。

同结构与联合一样,类也是一种用户定义的类型,它包括定义数据成员和定义成员函数两个方面,用数据成员来描述同类事物的属性,用成员函数来描述它们的行为。另外,用类定义变量(对象)也同用结构或联合定义变量一样,具有完全相同的语法格式,用类访问数据成员和成员函数,也同用结构变量或联合变量访问它们一样,是通过点操作符直接指明或通过箭头操作符间接指明来实现的。

在由C++提供的、允许用户使用的结构、联合和类类型中,它们的定义和使用具有完全相同的格式,上一章我们简单讨论了结构与联合中的数据成员以及它们的应用情况,这一章将以类类型为例深入讨论用户类型的定义、使用及有关知识。

### 8.1 类的定义

#### 8.1.1 类的定义格式

类定义的基本格式为:

```
class <类名> {<成员表>};
```

class为类类型定义的关键字,其后的<类名>为用户命名的标识符,以后可以利用它定义该类的变量以及引申的引用、指针和数组变量,<成员表>为该类包含的数据成员和成员函数的列表,每一个成员都具有一定的存取权限,或者称存取属性、访问权限、访问属性等,它由存取指明符关键字 public,private 或 protected 所指定。具体使用时,应在存取指明符后面加上一个冒号,使之与成员定义分开,此后的所有成员都具有该存取指明符所规定的存取权限,直到出现另一个存取指明符改变存取权限为止。若成员定义的前面没有使用存取指明符规定存取权限,则对于类成员来说隐含具有 private 访问属性,对于结构和联合成员来说隐含具有 public 访问属性。三个存取指明符的含义如下:

**public:** 公用(又称公有、公共)访问属性,成员可以为任意函数所使用。

**private:** 私有访问属性,成员只能为该类的成员函数和友元函数所使用。

**protected:** 保护访问属性,成员只能为该类的成员函数和友元函数以及派生类中的成员函数和友元函数所使用。关于友元函数和派生类的概念稍后介绍。

一般情况,按照面向对象的程序设计的要求,把类中的数据成员定义为私有的,这样只允许该类的成员函数访问,不允许该类以外的任何函数(包括具有全局作用域的一般函数和其他类中的具有类作用域的成员函数)访问,从而使类对象中的数据得到了隐藏和保护,不会受到外界有意或无意的破坏。

### 8.1.2 定义格式举例

```
(1) struct CA {  
    int a;  
    int b;  
} ax;
```

语句(1)定义了一个结构类型 CA,它包含有两个整数成员 a 和 b,由于没有显式规定其访问属性,所以采用隐含属性 **public**,即公用访问属性。利用该结构定义对象后,在对象作用域范围内,均可访问对象中的成员 a 和成员 b。例如,在该语句中同时定义了一个对象 ax,在 ax 作用域范围内,均可使用 ax.a 和 ax.b 作为表达式中的左值或右值。

```
(2) class CB {  
    int a;  
    int b;  
} bx;
```

语句(2)定义了一个类类型 CB,与语句(1)相同,它同样包含有两个整数成员 a 和 b,但由于没有显式规定其访问属性,所以采用隐含属性 **private**,即私有访问属性。利用该类定义对象后,在对象作用域范围内,均不能访问对象中的成员 a 和成员 b,若出现有进行访问的操作,则在程序编译时显示出“Cannot access private member declared in class ‘CB’”错误信息;表示不允许访问 CB 类中的私有成员,致使编译无法通过。因此对于含有私有成员类,外界只能够通过类中提供的公用成员函数间接访问其私有成员,若没有提供有关的公用成员函数则就无法访问了。在该语句中,既把 a 和 b 规定为私有成员,又没有提供访问它们的公用成员函数,所以无法访问它们,这表明该类的定义虽然语法正确,但没有实际意义。

```
(3) class CC {  
    int a;  
    public:  
    void Init(int aa)  
    {  
        a = aa;  
    }  
    int GetData()  
    {  
        return a;  
    }  
}
```

```
    } cx;
```

语句(3)定义了一个类类型 CC,它具有一个数据成员,即整数成员 a,其访问属性为私有的,该类具有两个访问属性为公用的成员函数,Init 成员函数能够把形参 aa 的值赋给成员变量 a,达到初始化 a 的目的。

注意:形参变量不能与成员变量重名,若重名只是把形参的值赋给形参变量,而不会赋给成员变量,因为在成员函数中,形参变量屏蔽了同名的成员变量。

类中另一个成员函数 GetData 能够返回成员变量 a 的值。该语句同时定义了一个类变量 cx,由于类成员 a 是私有的,所以无法用 cx.a 存取 cx 中的成员 a,但由于 Init 和 GetData 成员函数是公用的,所以通过 cx.Init(x)的调用可以把实参 x 的值传送给形参 aa,接着在执行函数体时由 aa 赋给 cx 中的成员变量 a,从而达到对 cx 的私有成员 a 赋值的目的;通过 cx.GetData()的调用能够返回 cx 中成员变量 a 的值,从而达到从 cx 的私有成员 a 中取值的目的。

```
(4) class CD {
    char * a;
    int b;
public:
    void Init(char* aa, int bb)
    {
        a = new char[strlen(aa) + 1];
        strcpy(a, aa);
        b = bb;
    }
    char* Geta() {return a;}
    int Getb() {return b;}
    void Output() {cout << a << ' ' << b << endl;}
} dx;
```

语句(4)定义了一个类类型 CD,它含有一个字符指针成员 a 和一个整数成员 b,这两个数据成员都是私有的,该类型同时包含有四个公用的成员函数,Init 函数给成员变量 a 和 b 赋值,并且 a 所指向的字符串存储空间是根据形参 aa 的长度动态分配产生的,Get 函数返回字符指针成员 a 的值,Getb 函数返回整数成员 b 的值,Output 函数输出字符指针成员 a 所指向的字符串和整数成员 b 的值。该语句同时定义了一个类变量 dx,假定利用 dx.Init("XXKWR",30)调用,则使得 dx.a 指向一个长度为 6 的动态分配的字符串空间,该空间存储的字符串为"XXKWR",该调用使得 dx.b 的值为 30;利用 dx.Geta()调用时,返回 dx.a 的值,由它可以得到所指向的字符串;利用 dx.Getb()调用时,返回 dx.b 的值;利用 dx.Output()调用时,在屏幕上显示出 dx.a 所指向的字符串和 dx.b 的值。

```
(5) class CE {
    private:
        int a,b;
        int getmax() {return (a > b ? a : b);}
    public:
        int c;
        void SetValue(int x1, int x2, int x3) {
```

```

        a = x1; b = x2; c = x3;
    }
    int GetMax() {
        int d = getmax();
        return (d > c ? d : c);
    }
} ex, *ep = &ex;

```

语句(5)定义了名为 CE 的类,它把整数成员 a 和 b 定义为私有的,把整数成员 c 定义为公用的,所以 a 和 b 只能由该类的成员函数访问,而 c 既可以为该类的成员函数访问,又可以为类外的任何函数访问。CE 类定义了一个私有成员函数 getmax,该函数返回成员变量 a 和 b 中的最大值,由于该函数是私有的,所以它只能为该类的成员函数调用,不能为类外的任何函数调用。CE 类同时定义了两个公用函数 SetValue 和 GetMax,前者为该类的三个数据成员赋值,后者返回三个数据成员中的最大值,该函数首先调用私有成员函数 getmax 得到 a 和 b 中的最大值并赋给 d,然后返回 d 和 c 中的大者,它就是 a, b, c 中的最大值。语句(5)同时定义了 CE 类的变量 ex 和指针变量 ep,并使 ep 初始化为 ex 的地址,这样 \*ep 与 ex、ep -> c 与 ex.c、ep -> SetValue(n1, n2, n3) 与 ex.SetValue(n1, n2, n3)、ep -> GetMax() 与 ex.GetMax() 等价,其中假定 n1, n2 和 n3 为三个整型实参变量。当系统执行 ex.GetMax() 调用时,首先调用私有函数 getmax(),返回 ex.a 和 ex.b 中的最大值,并把它赋给临时变量 d,接着返回 d 和 ex.c 中的最大值。

### 8.1.3 有关说明

1. 类中的成员函数和数据成员一样,都是在给定类中定义的,所有在外部调用成员函数同在外访问数据成员一样,必须带有所属类的对象名和成员操作符作为前缀,也就是说,必须通过对象(包括直接对象和指针对象)调用成员函数。由对象调用成员函数的过程就是对该对象进行操作的过程。当然外部能够访问的数据成员和成员函数都必须是类的公用成员。

2. 在一个类的成员函数中,能够直接使用该类定义的所有数据成员和成员函数,其使用表示也是抽象的,因为并没有将它们同任何一个类对象具体联系起来。当用类对象调用成员函数时,才给成员函数使用的成员赋予确定的含义,即把它们与具体对象联系起来。如 ex 为上述定义的 CE 类中的一个直接对象,进行 ex.SetValue(2, 3, 4) 调用时,该成员函数中使用的 a, b, c 才被具体确定为对象 ex 中的 ex.a, ex.b 和 ex.c 成员,调用执行结束后,ex 中三个数据成员的值分别为 2, 3 和 4;若 ey 为 CE 类中的另一个对象,则进行 ey.SetValue(2, 3, 4) 调用时,该成员函数中使用的 a, b, c 确定为 ey 中的 ey.a, ey.b 和 ey.c 成员,调用执行结束后,ey 中三个数据成员的值分别为 2, 3 和 4。又如进行 ex.GetMax() 调用时,该成员函数中使用的 getmax() 被确定为 ex.getmax(),进行 ex.getmax() 调用时,这个成员函数中使用的 a 和 b 被确定为 ex.a 和 ex.b,执行后返回它们的最大值并赋给 ex.GetMax() 中的局部变量 d,接着返回 d 和 ex.c 中的最大值。

3. 一个类的每个成员函数中都隐含有一个所属类的指针参数,其名字由系统规定为 this,它是一个 C++ 关键字。当由一个对象调用一个成员函数时,除了把实参传送给成员函

数中被说明的形参外,还同时把该直接对象的地址或指针对象的值传送给成员函数中隐含说明的指针形参 `this`,执行成员函数时对数据成员的访问实际上是对 `this` 指针所指对象中数据成员的访问,对成员函数的调用实际上是对 `this` 指针所指对象中成员函数的调用。在成员函数中可以利用 `this` 指针做任何操作,如将所使用的成员名前加上 `this->` 限定符,则与不加完全相同,又如使用 `this` 和 `*this` 分别表示调用该函数的类对象指针和类对象本身。

4. 成员函数可以在类中给出定义,在上述格式中都是如此,也可以使成员函数的声明与定义分开,即只在类中给出声明,而在类外给出完整的定义。

**注意:**成员函数在类外定义时,函数名前面必须带有所属类的类名和类区分符`::`,以指定该成员函数的所在类,否则它将是一个具有全局作用域的普通函数。

例如,对于上述定义的 `CE` 类,若把它的所有成员函数都放在类外定义,则重写 `CE` 类如下:

```
class CE {
private:
    int a,b;
    int getmax(); // 成员函数声明
public:
    int c;
    void SetValue(int x1, int x2, int x3);
        // 成员函数声明,每个形参名可以省略
    int GetMax(); // 成员函数声明
} ex, *ep = &ex;

int CE::getmax() { // getmax() 的类外定义
    return (a > b ? a : b);
}

void CE::SetValue(int x1, int x2, int x3)
{ // SetValue(int, int, int) 的类外定义
    a = x1; b = x2; c = x3;
}

int CE::GetMax() { // GetMax() 的类外定义
    int d = getmax();
    return (d > c ? d : c);
}
```

5. 成员函数同普通函数一样,也可以通过使用 `inline` 关键字把它定义为内联函数。函数被定义为内联后,调用执行时可能具有更快的速度,一般只把简单的函数定义为内联的。另外,若成员函数是在类中定义的,则隐含为内联的,就如同使用了 `inline` 关键字一样,若成员函数是在类外定义的,则 `inline` 关键字加到函数声明或函数定义上都可以规定它是内联的。如将 `CE::GetMax()` 成员函数规定为内联的,则如下所示。

```
inline int CE::GetMax() { // GetMax() 的类外定义
    int d = getmax();
    return (d > c ? d : c);
}
```



)

6. 同普通函数一样,每个成员函数中说明的形参也允许带有默认值,带有默认值的形参必须放在参数表的尾部,而不带默认值的形参必须放在参数表的首部,不允许它们相互混淆。若成员函数的声明和定义是分开的,则参数的默认值只能在函数声明中给出。利用对象调用具有默认值的成员函数时,对应默认值的实参可以省略,当省略时则形参采用默认值,不省略时从对应实参中取得值。例如,对于上述 CE 类中声明的 SetValue(int, int, int)成员函数,假定第 2 和第 3 个形参的默认值为 0 和 1,则该成员函数的声明和定义如下:

```
void SetValue(int x1, int x2 = 0, int x3 = 1);
void CE::SetValue(int x1, int x2, int x3)
{
    a = x1; b = x2; c = x3;
}
```

例如当进行 ex.SetValue(4,5,7)调用时,ex.a,ex.b 和 ex.c 分别被赋值为 4,5 和 7;当进行 ex.SetValue(4,5)调用时,ex 中的三个数据成员分别被赋值为 4,5 和 1;当进行 ex.SetValue(4)调用时,又分别被赋值为 4,0 和 1。

7. 每个类类型的大小,即由该类定义的对象所占用存储空间的字节数等于所有数据成员所占用存储字节数的总和。每个类对象中只保留有数据成员的存储空间,不保留有成员函数的存储空间。当由一个对象调用一个成员函数时,将从所属类中得到编译后的该成员函数执行代码的入口地址,然后把实参的值或引用传送给形参,把对象的地址传送给 this 指针,再接着执行函数体,执行结束后返回到调用前的位置。例如 CE 类的大小为 12,即等于所含的三个整数成员的大小之和。

8. 同普通函数一样,在类中的成员函数也允许函数名重载和操作符重载。当一个类中的成员函数名相同,但参数个数、参数类型、参数顺序不完全相同时则称为重载成员函数。利用 C++ 中已经定义的操作符,可以按照用户的要求在类上重新定义出操作符函数,称此为操作符重载成员函数。

注意:当定义重载双目操作符的成员函数时,则参数表中只需要定义一个参数,该参数是进行双目运算的第二个参数,而第一个参数为调用该函数的对象,当定义重载单目操作符的成员函数时,则参数表为空(除重载后缀加或减需要一个整型参数外),单目运算的参数为调用该函数的对象。

假定用类来实现分数的定义和有关运算,则整个程序如下:

```
#include <iostream.h>
class Fraction { // 定义分数类
    int nume; // 定义分子
    int deno; // 定义分母
public:
    void FranSimp() { // 把 *this 化简为最简分数
        int m, n, r;
        m = nume; n = deno;
        // 把 *this 分数的分子和分母分别赋给 m 和 n 作为其初值
        r = m % n; // 将 m 整除以 n 的余数赋给 r 作为其初值
```

```

        while(r != 0) {
            m = n; n = r;
            r = m % n;
        }
        if(n != 1) { // 化简 *this 分数的分子和分母
            nume /= n;
            deno /= n;
        }
        if(deno < 0) { // 使 *this 分数的分母转为正
            nume = -nume;
            deno = -deno;
        }
    }

    void InitFracton() { // 置分数的分子和分母分别为 0 和 1
        nume = 0;
        deno = 1;
    }

    void InitFracton(int n, int d)
    { // 置分数的分子和分母分别为 n 和 d
        nume = n;
        deno = d;
    }

    Fracton operator + (Fracton& x)
    { // 返回两个分数 *this 和 x 之和
        Fracton c; // 用 c 保存求和结果
        c.nume = nume * x.deno + deno * x.nume; // 计算结果分数的分子
        c.deno = deno * x.deno; // 计算结果分数的分母
        c.FracSimp(); // 对结果分数进行简化处理
        return c; // 返回结果分数
    }

    int operator == (Fracton& x)
    { // 若分数 *this 和 x 的值相等则返回 1, 否则返回 0
        return nume * x.deno == deno * x.nume;
    }

    Fracton& operator ++ ()
    { // 先使分数增 1, 然后返回它的引用
        nume += deno;
        return *this;
    }

    Fracton operator ++ (int)
    { // 使分数增 1, 但返回的是分数的原值
        Fracton y = *this;
        nume += deno;
        return y;
    }
}

```

```

void FranOutput()
{ // 输出一个分数
    cout << nume << '/' << deno << endl;
}
}; // 类定义结束

void main()
{
    Fraction a,b,c; // 定义 a,b,c 三个分数类对象
    a.InitFraction(7,12); // 假定给 a 的分子和分母赋值为 7 和 12
    b.InitFraction(-3,8); // 假定给 b 的分子和分母赋值为 -3 和 8
    c.InitFraction(); // c 中的分子和分母被赋值为 0 和 1
    c = a + b; // a + b 等价于 a.operator + (b)
    cout << "a: "; a.FranOutput(); // 输出分数 a
    cout << "b: "; b.FranOutput(); // 输出分数 b
    cout << "c: "; c.FranOutput(); // 输出分数 c
    if(a == b) // a == b 等价于 a.operator == (b)
        cout << "a == b" << endl;
    else
        cout << "a != b" << endl;
    if(c == a + b) // c == a + b 等价于 c.operator == (a + b)
        cout << "c == a + b" << endl;
    else
        cout << "c != a + b" << endl;
    c = ++a; // ++a 等价于 a.operator ++ ()
    cout << "a: "; a.FranOutput();
    cout << "c: "; c.FranOutput();
    c = a ++; // a ++ 等价于 a.operator ++ (1)
    cout << "a: "; a.FranOutput();
    cout << "c: "; c.FranOutput();
}

```

该程序的运行结果为:

```

a: 7/12
b: -3/8
c: 5/24
a != b
c == a + b
a: 19/12
c: 19/12
a: 31/12
c: 19/12

```

在程序所定义的分數 Fraction 类中,出现了 InitFraction 成员函数的重载,这种重载也可以用下面的带参数默认值的函数来代替,但略有区别,留给读者分析。

```

void InitFraction(int n=0, int d=1)
{ // 置分数的分子和分母分别 n 和 d
    nume = n;
    deno = d;
}

```

```
}
```

在类 `Fracton` 的定义中,没有给出分数的输入输出操作的提取操作符 `>>` 和插入操作符 `<<` 的重载函数,因为它需要用到类的友元函数,这留在稍后介绍。

## 8.2 构造函数

构造函数和析构函数属于类中的成员函数,由于它们具有特殊和重要的作用,所以在这一节和下一节进行专门讨论。

构造函数与所在的类具有相同的名字,并且不带任何返回类型,也不需要返回任何值,函数的参数表和函数体由用户根据需要编写,是否为一个类建立重载的多个构造函数,也由用户根据使用类的需要而定。建立构造函数的作用是为类对象中的数据成员赋初值,有时还要为其中的指针数据成员动态分配所指向的存储空间。构造函数将由建立所属类的对象时自动调用,完成给对象赋初值的任务,而不允许像其他成员函数那样由用户直接调用。由于构造函数是在建立对象时自动调用的,而建立对象都是在类外进行的,所以必须把每个构造函数定义为类的公用成员。

### 8.2.1 无参构造函数和带参构造函数

下面通过具体例子来说明无参构造函数和带参构造函数的定义和使用。

例 1. 假定定义如下的数组类 `Array`,该类包含有两个构造函数,一个置数组指针 `a` 的初值为空,置数组长度 `n` 的初值为 0;另一个置数组指针 `a` 初始指向形参数组 `aa`,置数组长度 `n` 的初值为形参 `len` 的值。

```
class Array // 定义数组类
{
    int * a; // 定义指向整型数组的指针
    int n; // 定义数组长度
public:
    Array() {a=NULL; n=0;} // 无参构造函数,构造空数组
    Array(int aa[], int len)
    { // 带参构造函数,构造指向形参 aa 的数组
        if(len <= 0) {
            cerr << "n of value is invalid!" << endl;
            // n 的值无效
            exit(1);
        }
        n = len; // 给 n 赋初值为 len
        a = aa; // 给 a 赋初值为形参数组的地址
    }
    int Length() { // 返回数组的长度
        return n;
    }
    bool IsExist(int x) { // 判断 x 是否存在于数组 a 中
```

```

        for(int i=0; i<n; i++)
            if(a[i] == x) return true;
        return false;
    }
    int MaxValue() { // 返回数组 a 中的最大值
        if(n == 0) {
            cerr << "array a is empty!" << endl;
            exit(1);
        }
        int x=a[0];
        for(int i=1; i<n; i++)
            if(a[i]>x) x=a[i];
        return x;
    }
    void Sort(); // 按升序排列数组 a 中的 n 个元素,
                // 此为函数声明, 函数定义在类外
    void Output() { // 依次输出数组 a 中的 n 个元素值
        for(int i=0; i<n; i++)
            cout << a[i] << ' ';
        cout << endl;
    }
}; // 类定义结束

void Array::Sort() { // 成员函数的类外定义
    for(int i=1; i<=n-1; i++) {
        int k=i-1;
        for(int j=i; j<=n-1; j++)
            if(a[j]<a[k]) k=j;
        if(k!=i-1) {
            int x=a[i-1]; a[i-1]=a[k]; a[k]=x;
        }
    }
}

```

在上述 Array 类定义中,除两个构造函数外,还定义了五个公用成员函数,Length 函数返回数组长度,即数组中的元素个数;IsExist 函数判断形参 x 是否存在于数组 a 中,若 x 的值与数组 a 中某一个元素的值相等则返回“真”表示存在,否则返回“假”表示不存在;MaxValue 函数返回数组 a 中的最大值;Sort 函数对数组 a 中的 n 个整数元素,采用选择排序的方法按升序进行排序;Output 函数依次输出数组 a 中 n 个元素的值。

结合下面使用 Array 类的主函数,分析构造函数的调用情况。

```

void main()
{
    int a[] = {50,24,67,82,44,69}; // 定义数组 a 并初始化
    Array r1,r2(a,6); // 定义数组类对象 r1 和 r2
    cout << "r1.Length() = " << r1.Length();
        // 输出 r1 中的数组长度
    cout << ", r2.Length() = " << r2.Length() << endl;
        // 输出 r2 中的数组长度
    r1 = r2; // r2 赋值给 r1
}

```

```

cout << "before sorting r2.a: "; r2.Output();
    // 输出排序前 r2 中的数组值
r2.Sort();
    // 对 r2 中的数组排序
cout << "after sorting r2.a: "; r2.Output();
    // 输出排序后 r2 中的数组值
cout << "r1.a: "; r1.Output();
    // 输出 r1 中的数组值
cout << "r1.Length() = " << r1.Length() << endl;
    // 输出 r1 中数组的长度
cout << "r1.MaxValue() = " << r1.MaxValue() << endl;
    // 输出 r1 中数组的最大值
}

```

在主函数中,第一条语句定义了一个整型数组 `a`,并初始化为 6 个数据元素。第二条语句定义了两个 `Array` 类对象,定义 `r1` 时自动调用无参构造函数,给 `r1` 的整数指针 `a` 赋初值为 `NULL`,整数成员 `n` 赋初值为 0,表示建立了一个空数组;定义 `r2` 时自动调用带参构造函数,把 `r2` 后面括号内的实参值对应传送给形参变量 `aa` 和 `len`,接着执行构造函数时再把它们的值分别赋给 `r2.a` 和 `r2.n`,使得 `r2.a` 指向上述定义的整型数组 `a`,`r2.n` 的值为 6,当对 `r2.a` 数组进行操作时实际上就是对上述定义的整型数组 `a` 的操作。第三和第四条语句分别显示出 `r1` 和 `r2` 中的数组长度。第五条语句把 `r2` 的值赋给 `r1`,使 `r1` 与 `r2` 相同,即 `r1.a` 也同时指向第 1 条语句定义的整型数组 `a`,`r1.n` 的值也同样为 6。第六条语句输出 `r2` 中的数组值。第七条语句对 `r2` 中的数组进行排序。第八条语句输出排序后的 `r2` 中的数组值。第九至十一条语句分别输出 `r1` 中的数组值、数组长度和数组中的最大值。由于 `r1.a` 和 `r2.a` 同时指向主函数中的数组 `a`,所以对其中任一个数组成员的操作也是对另一个数组成员的操作。主函数运行结果如下:

```

r1.Length() = 0, r2.Length() = 6
before sorting r2.a: 50 24 67 82 44 69
after sorting r2.a: 24 44 50 67 69 82
r1.a: 24 44 50 67 69 82
r1.Length() = 6
r1.MaxValue() = 82

```

当定义一个类时,若没有定义构造函数,则系统隐含定义一个无参构造函数,该构造函数的函数体也为空,调用系统所给的构造函数不会执行任何操作。如对于上面定义的 `Array` 类,若没有定义任何构造函数,则系统隐含定义的构造函数为:

```
Array() {}
```

当用类类型定义一个类对象时,若需要它自动调用无参构造函数对其进行初始化,则只需给出对象名,若需要它自动调用带参构造函数对其进行初始化,则给出的对象名后必须带有用圆括号括起来的实参表。

类定义中的无参构造函数可能是系统定义的(用户未定义任何构造函数时),也可能是用户定义的,但带参构造函数必定是由用户定义的。

类对象定义所带的参数表中若只有一个参数时,也可以用赋值号代替圆括号。如类对

象定义中的  $x(5)$  可表示为  $x=5$ , 其含义相同。

当由动态分配一个类对象时, 对该对象进行初始化所需要的参数表应放在 `new` 操作符后的类名的后面, 当然调用无参构造函数时就不带参数表。如要动态产生两个 `Array` 类的对象, 使一个对象如同上述主函数中的 `r1`, 另一个对象如同 `r2`, 并假定指向它们的指针分别为 `s1` 和 `s2`, 则进行动态分配类对象的语句如下:

```
Array * s1 = new Array;  
Array * s2 = new Array(a, 6);
```

执行第一条语句时, 首先动态分配一个具有 `Array` 类型大小的存储空间, 并把它的首地址赋给 `s1`, 接着自动调用 `Array` 类的无参构造函数, 使动态对象 `*s1` 的 `a` 成员的空, 即 `s1 -> a = NULL`, 使 `*s1` 的 `n` 成员的空, 即 `s1 -> n = 0`。执行第二条语句时, 首先动态分配一个具有 `Array` 类型大小的存储空间, 并把它的首地址赋给 `s2`, 接着自动调用 `Array` 类的带参构造函数, 把 `a` 的值传送给形参 `aa`, `6` 传送给形参 `len`, 执行函数体时把 `aa` 的值赋给动态对象 `*s2` 的 `a` 成员, 即赋给 `s2 -> a`, 把 `len` 的值赋给 `*s2` 的 `n` 成员, 即赋给 `s2 -> n`。

下面的主函数与上述的主函数具有完全相同的功能, 除了输出的字符串提示信息外, 其输出结果是相同的。

```
void main()  
{  
    int a[] = {50, 24, 67, 82, 44, 69}; // 定义数组 a 并初始化  
    Array * r1 = new Array; // 定义一个动态对象 *r1  
    Array * r2 = new Array(a, 6); // 定义一个动态对象 *r2  
    cout << "r1 -> Length() = " << r1 -> Length();  
    // 输出 *r1 中的数组长度  
    cout << ", r2 -> Length() = " << r2 -> Length() << endl;  
    // 输出 *r2 中的数组长度  
    r1 = r2; // r2 赋值给 r1, 使 r1 也指向 r2 所指向的对象  
    cout << "before sorting r2 -> a: "; r2 -> Output();  
    // 输出排序前 *r2 中的数组值  
    r2 -> Sort();  
    // 对 *r2 中的数组排序  
    cout << "after sorting r2 -> a: "; r2 -> Output();  
    // 输出排序后 *r2 中的数组值  
    cout << "r1 -> a: "; r1 -> Output();  
    // 输出 *r1 中的数组值  
    cout << "r1 -> Length() = " << r1 -> Length() << endl;  
    // 输出 *r1 中数组的长度  
    cout << "r1 -> MaxValue() = " << r1 -> MaxValue() << endl;  
    // 输出 *r1 中数组的最大值  
}
```

这个主函数的运行结果如下:

```
r1 -> Length() = 0, r2 -> Length() = 6  
before sorting r2 -> a: 50 24 67 82 44 69  
after sorting r2 -> a: 24 44 50 67 69 82  
r1 -> a: 24 44 50 67 69 82  
r1 -> Length() = 6
```

```
r1 -> MaxValue() = 82
```

例2. 在例1定义的数组类 Array 中,使用的数组是形参 aa 指针所指向的数组,而不是由该类自己建立的数组,这样对该类中数组的操作实际上是在实参数组中进行的。为了使类中的数据具有独立性,就必须定义有自己的存储空间,而不是使用类外的存储空间。假定对 Array 类定义自己的非动态分配的数组空间,则该类的定义如下:

```
class Array // 定义数组类
{
    int a[10]; // 定义数组最大空间,假定最多保存 10 个整数元素
    int n; // 定义数组中当前所存的数据个数,即当前数组长度
public:
    Array() {n=0;} // 构造空数组,即当前数组长度为 0 的数组
    Array(int aa[], int len)
    { // 利用 aa 数组和 len 初始化数组 a 和 n
        if(len <= 0) {
            cerr << "n of value is invalid!" << endl;
            // n 的值无效
            exit(1);
        }
        n=len; // 给 n 赋初值为 len
        for(int i=0; i<n; i++)
            a[i]=aa[i]; // 把 aa 的每个元素值赋给 a 数组对应元素中,
                        // 此后类中数组内容的变化与实参数组无关
    }
    // 类中的其他五个成员函数与例 1 完全相同,在此不重写
};
```

假定仍使用上述第 1 个主函数,则程序运行结果如下:

```
r1.Length() = 0, r2.Length() = 6
before sorting r2.a: 50 24 67 82 44 69
after sorting r2.a: 24 44 50 67 69 82
r1.a: 50 24 67 82 44 69
r1.Length() = 6
r1.MaxValue() = 82
```

由于 r2 对 r1 赋值后,对 r2 中的数组进行了排序,而 r1 中的数组内容不变,所以显示的 r1 中数组的内容为原 r2 中数组的内容。

例3. 类中的数组采用固定(即非动态)的存储空间分配,它需要分配应用中最大的数组空间,但这对于一般的应用又造成存储空间的浪费,所以通常对类中使用的数组进行动态分配,这样能够根据每个对象的实际需要分配合适的存储空间,使得类的定义既具有灵活性和适应性,又避免了存储空间的浪费。假定对 Array 类中的数组采用动态分配数组空间,则该类的定义如下:

```
class Array // 定义数组类
{
    int * a; // 定义数组指针,用以指向一个动态数组空间
    int n; // 定义数组长度
public:
    • 244 •
```



```

Array() {a=NULL; n=0;} // 构造空数组
Array(int aa[], int len)
{ // 利用 aa 数组构造数组长度为 len 的数组 a
    if(len <= 0) {
        cerr << "n of value is invalid!" << endl;
        // n 的值无效
        exit(1);
    }
    n=len; // 给 n 赋初值为 len
    a=new int[n]; // 建立动态数组
    for(int i=0; i<n; i++)
        a[i]=aa[i]; // 把 aa 的每个元素值赋给 a 数组对应元素中
}
// 类中的其他五个成员函数与例 1 完全相同,在此不重写
};

```

### 8.2.2 拷贝构造函数

类中的构造函数当有并且只有一个所属类的引用参数时,则称此构造函数为拷贝构造函数或复制构造函数。当用一个类对象初始化同类的另一个对象时需要调用相应的拷贝构造函数。如假定  $a$  是类  $A$  的一个对象,则当定义类  $A$  的另一个对象  $b(a)$  时(此时的  $b(a)$  也可表示为  $b=a$ ),就会自动调用类  $A$  的拷贝构造函数,把  $a$  作为初值来初始化  $b$  对象。若一个类中不带有拷贝构造函数,则系统为该隐定义一个拷贝构造函数。如假定类  $X$  中不带有拷贝构造函数,则系统为类  $X$  隐含定义的拷贝构造函数为:

```

X(X&x) // 假定引用形参用 x 表示
{
    *this=x; // 将引用形参的值赋给被初始化的对象
}

```

系统给定的拷贝构造函数把初始化对象的值赋给被初始化的对象中,也就是说,把用于初始化的对象中每个字节的内容原原本本地拷贝到被初始化对象的对应字节中,使得被初始化对象的每个成员值与初始化对象的每个成员值完全相同。

在一个类中,若存在指针数据成员指向动态分配的存储空间,采用系统隐含定义的拷贝构造函数用一个对象初始化另一个对象后,这两个对象的相应指针数据成员将指向同一个动态分配的存储空间,当一个对象中的动态存储空间由于某种原因使用 `delete` 操作释放给系统之后,另一个对象中的相应指针成员仍指向这个已被释放掉的存储空间,如果再通过它访问这个存储空间时,将是无效和非法的。所以为了避免这种情况的发生,需要用户定义自己的拷贝构造函数,在这个函数中,要为被初始化对象动态分配与初始化对象中相应指针成员所指向的同样大小的动态存储空间,并让指针成员指向它,接着进行动态存储空间之间数据的拷贝。这样,两个对象中相应指针成员就各自指向了不同的动态存储空间,一个对象中动态分配的存储空间的释放绝不会影响另一个对象中相应动态存储空间的存在,因此不会出现采用系统隐含提供的拷贝构造函数所带来的问题。

对于上述定义的具有动态数组空间的 `Array` 类,应为它提供的拷贝构造函数如下:

```

Array::Array(Array& x)
{
    // 将 x 中 n 成员的值赋给 *this 中的 n 成员
    n = x.n;
    // 动态分配大小为 n 的整型数组并由 *this 中的 a 成员所指向
    a = new int[n];
    // 把 x.a 数组中每个元素值赋给 this -> a 数组的对应元素中
    for(int i = 0; i < n; i++)
        a[i] = x.a[i];
}

```

假定通过下面的语句定义 Array 类的两个对象 r1 和 r2。

```
Array r2(a,6), r1(r2);
```

当C++系统执行这条语句时,首先为 r2 分配大小为 8 个字节的存储空间,接着调用带参构造函数,把实参 6 通过形参 len 赋给 r2.n 中,把实参 a 数组中的 6 个元素值通过指针形参 aa 赋给 r2.a 指针所指向的动态存储空间中;然后为 r1 分配大小为 8 个字节的存储空间,接着调用拷贝构造函数,把 x 作为 r2 的别名,通过此别名把 r2.n 的值赋给 r1.n 中,把 r2.a 数组中的元素值赋给 r1.a 指针所指向的动态存储空间中,此时虽然 r1.a 和 r2.a 数组的值相同,但它们所对应的存储空间不同,即 r1.a 和 r2.a 的指针值不同。图 8-1(a)和 8-1(b)分别为 r2 和 r1 的存储映像,即存储状态示意图。

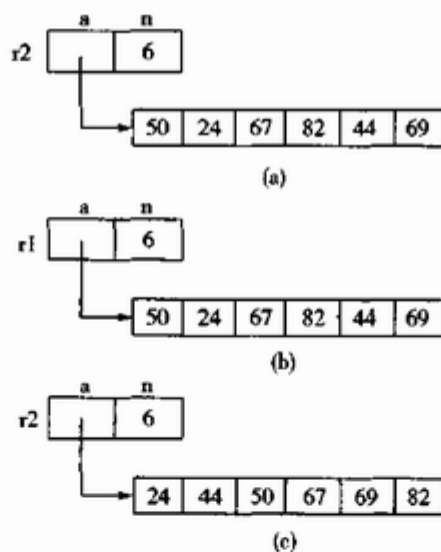


图 8-1 r2 和 r1 的存储映像

当对 r2 中数组的元素值按升序排序后, r2 的存储映像如图 8-1(c)所示。由于没有对 r1 中的数组进行排序操作,所以 r1 的存储映像保持不变。

在程序执行中,除了用一个已定义类对象初始化另一个被定义的同类对象时自动调用拷贝构造函数外,还在类的实参值传送给对应值参的过程中,以及在函数中将类对象作为值返回时也将自动调用拷贝构造函数。

### 8.2.3 赋值重载函数

在一个类中,若存在指针成员指向动态分配的存储空间的情况,则进行类对象赋值时,同样也是把一个对象中指针成员的值赋给了另一个对象的相应指针成员中,使得它们共同指向了同一个动态存储空间。当一个对象中指针成员所指向的动态存储空间被释放后,致使另一个对象中相应指针成员指向了一个无效的存储空间。为了避免这种现象的发生,同样可以定义一个赋值操作符重载函数。该函数与拷贝构造函数类似,它有并且只有一个所属类的参数,不过此时可以为引用,也可以为值参,该函数应返回被赋值对象的引用,以便能够赋值给同类的其他对象,即可以使赋值号连用。在该函数的函数体内应首先删除被赋值对象的指针成员所指向的动态存储空间,接着进行与拷贝构造函数相同的操作,最后返回被赋值的对象。

对于上述定义的 Array 类,应为它提供的赋值重载函数如下:

```
Array& Array::operator = (Array& x)
{
    delete []a;
    n = x.n;
    a = new int[n];
    for(int i = 0; i < n; i++)
        a[i] = x.a[i]; // 把 x.a 中每个元素值赋给 a 数组对应元素中
    return *this;
}
```

当进行类对象赋值时,若在该类中存在有赋值重载函数,则调用该函数完成赋值操作,否则就采用系统默认的赋值方法,把赋值号右边对象的值拷贝到赋值号左边的对象中。假定 r1 和 r2 为两个同类对象,则执行赋值表达式 r1 = r2 时,若相应的赋值重载函数存在,就调用该函数完成 r2 向 r1 赋值的操作,该表达式 r1 = r2 等价于 r1.operator=(r2)表示。

### 8.2.4 构造函数中的初始化表

在构造函数中给数据成员赋值时,可以采用两种格式,一种是在函数体中使用赋值语句把表达式的值赋给成员变量,以上讨论的情况均如此;第二种是在形参表后和函数体之前使用初始化表(又称初值表)赋值,并且初始化表同形参表之间必须用冒号分开,初始化表中用逗号分开的每一项用于给一个成员变量赋值,每一项的格式为“成员变量名(初值表达式)”。例如对于 Array 类中的无参构造函数,若采用初始化表为每个成员变量赋值,则定义如下:

```
Array(): a(NULL), n(0) {}
```

当然也可以一部分成员采用初始化表赋值,另一部分成员采用函数体赋值。如下面定义 Array 类的无参构造函数也是正确的。

```
Array(): a(NULL) {n = 0;}
```

对于 Array 类中的带参构造函数,在确保形参 len 大于等于 1 的情况下,可以采用如下

定义:

```
Array(int aa[], int len): n(len), a(new int[len])
{
    for(int i=0; i<n; i++)
        a[i] = aa[i];
}
```

当带有初始化表的构造函数执行时,首先执行初始化表,然后才执行函数体。在执行初始化表时,不管各项的排列次序如何,它都将按照类中数据成员定义的先后次序给数据成员赋初值。如在执行上面的构造函数时,不管初始化表中数据成员 *a* 和 *n* 项的先后次序如何,都将先对 *a* 赋初值,然后再对 *n* 赋初值。当然在执行函数体为数据成员赋初值时,是按照语句排列的顺序从上到下(同一行从左到右)执行的。

### 8.3 析构函数

析构函数的名字也与类名相同,不过应在函数名前加上波折号(~),以示同构造函数的区别。析构函数不允许带任何参数,并且也不允许带有返回类型。析构函数可以同其他成员函数一样由对象调用,但通常同构造函数一样是由系统执行程序时自动调用的,但调用时机正好相反,构造函数是在对象生成时调用的,而析构函数是在对象撤消时调用的,并且调用执行后才真正撤消对象。对于定义的非动态对象,当程序执行离开它的作用域时将自动被撤消,对于动态对象,只有当对其执行 *delete* 操作时才撤消,否则不会被自动撤消。由于撤消对象是在类外进行的,而析构函数是在撤消对象时调用的,所以必须把析构函数定义为公用成员函数。

当用户没有给一个类定义析构函数时,系统隐含给这个类定义一个析构函数,该函数的函数体为空,所以当自动调用系统给定的析构函数时不会执行任何操作。例如,设一个类为 *X*,当用户没有定义析构函数时,系统为该隐类隐含定义的析构函数为:

```
~X() {}
```

由于析构函数是在对象撤消时被自动调用的,所以通常利用析构函数删除对象中由指针成员所指向的动态分配的存储空间,当类中不使用动态存储空间时,则通常不需要定义析构函数。

对于 8.2.1 小节例 1 和例 2 中定义的 *Array* 类,则不需要专门定义析构函数,而对于例 3 中定义的 *Array* 类,由于使用了动态存储空间,所以需要专门定义析构函数,用来释放由指针成员 *a* 所指向的动态数组空间。该类的析构函数定义如下:

```
Array() { // 析构函数
    delete []a;
    cout << "destructor" << endl;
    // 此条语句是让用户从屏幕输出中能够看到析构函数被调用的情况
}
```

下面给出含有无参构造函数、带参构造函数、拷贝构造函数、析构函数和赋值重载函数

的 Array 类的定义:

```
class Array
{
    int * a;
    int n;
public:
    Array(): a(NULL), n(0) {}
    Array(int aa[], int len): a(new int[len]), n(len)
    {
        for(int i = 0; i < n; i++)
            a[i] = aa[i];
    }
    Array(Array& x); // 拷贝构造函数的声明
    Array& operator = (Array& x); // 赋值重载函数的声明
    ~Array() { // 析构函数
        delete []a;
        cout << "destructor" << endl;
    }
    void Exchange()
    { // 倒序数组中的元素值
        for(int i = 0; i < n/2; i++) {
            int x = a[i]; a[i] = a[n-i-1];
            a[n-i-1] = x;
        }
    }
    // 类中的其他五个成员函数与 8.2.1 小节例 1 中完全相同, 在此不重写
};

Array::Array(Array& x)
{
    n = x.n;
    a = new int[n];
    for(int i = 0; i < n; i++)
        a[i] = x.a[i]; // 把 x.a 中每个元素值赋给 a 数组对应元素中
}

Array& Array::operator = (Array& x)
{
    delete []a;
    n = x.n;
    a = new int[n];
    for(int i = 0; i < n; i++)
        a[i] = x.a[i]; // 把 x.a 中每个元素值赋给 a 数组对应元素中
    return *this;
}
```

下面的主函数使用了 Array 类。

```
void main()
{
    int a[] = {50, 24, 67, 82, 44, 69};
```

```

Array r(a,6); // 定义类对象 r 并调用带参构造函数
Array * s = new Array(r); // 定义动态对象 *s 并用 r 初始化
Array * t = new Array; // 定义动态对象 *t 并调用无参构造函数
*t = *s; // 调用赋值构造函数把 *s 赋值给 *t
r.Sort(); // 排序 r.a[] 数组中的元素值
t -> Exchange(); // 倒序 t -> a[] 数组中的元素值
cout << "r.a[]: "; r.Output();
// 输出排序后 r 中的数组值
cout << "s -> a[]: "; s -> Output();
// 输出 s 所指对象中的数组值
cout << "t -> a[]: "; t -> Output();
// 输出倒序后 t 所指对象中的数组值
cout << "删除 s 所指的动态对象 *s:" << endl;
delete s; // 自动调用析构函数
cout << "删除 t 所指的动态对象 *t:" << endl;
delete t; // 自动调用析构函数
cout << "程序结束:" << endl;
// 主函数结束之前撤消非动态对象 r 时自动调用析构函数
)

```

该程序运行结果如下:

```

r.a[]: 24 44 50 67 69 82
s -> a[]: 50 24 67 82 44 69
t -> a[]: 69 44 82 67 24 50
删除 s 所指的动态对象 *s:
destructor
删除 t 所指的动态对象 *t:
destructor
程序结束:
destructor

```

## 8.4 友元函数和友元类

在一个类中,类对象的私有成员只能由该类的成员函数访问,外部定义的普通函数和其他类中定义的成员函数都不得访问,这些外部函数只能通过该类提供的公用成员函数进行访问,这样有利于数据的封装、隐藏和保护,符合面向对象程序设计的要求。但当一个函数或一个类与另一个类关系比较密切,即它们需要经常访问另一个类中的数据时,由于不能直接访问另一个类的私有数据成员,必须通过调用公用成员函数来实现,这将带来很低的访问效率,即访问速度很慢。为了提高访问效率,C++ 允许在一个类中把外部的有关函数或类声明(或称为说明、宣布等)为它的友元函数或友元类,被声明为一个类的友元函数或友元类具有直接访问该类的私有成员的特权。直接访问私有成员比通过调用成员函数访问私有成员具有更高的效率。但这将破坏数据的封装性,所以要有限制地使用,不能乱用。最好只在关系比较密切的类与类之间、类与函数之间使用。

下面通过例子来说明如何在类中声明友元函数和友元类,以及它们访问类中私有成员

的情况。

例 1. 下面的程序定义了一个分数类,该类把进行分数输入和输出的提取和插入操作符重载函数声明为它的友元函数,当然这两个友元函数都是类外的普通函数。

```
#include <iostream.h>
#include <stdlib.h>
class Fraction; // 分数类的不完整定义,
// 为下面两个函数声明使用该类作参数类型提供根据
istream& operator >> (istream&, Fraction&);
ostream& operator << (ostream&, Fraction&);

class Fraction { // 定义分数类
    int nume; // 定义分子
    int deno; // 定义分母
public:
    friend istream& operator >> (istream&, Fraction&);
    // 声明进行分数输入的提取操作符重载函数为该类的友元函数
    friend ostream& operator << (ostream&, Fraction&);
    // 声明进行分数输出的插入操作符重载函数为该类的友元函数
    void Fransimp(); // 函数声明 具体函数定义如 8.1 节给出的那样
    Fraction()
    { // 无参构造函数,置分数的分子和分母分别为 0 和 1
        nume = 0;
        deno = 1;
    }
    Fraction(int n, int d)
    { // 带参构造函数,置分数的分子和分母分别为 n 和 d
        nume = n;
        deno = d;
    }
    void InNume(int n) { // 重写分子 nume 的值
        nume = n;
    }
    void InDeno(int d) { // 重写分母 deno 的值
        deno = d;
    }
    int GetNume() { // 取出分子 nume 的值
        return nume;
    }
    int GetDeno() { // 取出分母 deno 的值
        return deno;
    }

    Fraction operator + (Fraction& x);
    // 函数声明 具体函数定义如 8.1 节给出的那样
    int operator == (Fraction& x);
    // 函数声明 具体函数定义如 8.1 节给出的那样
    int operator > (Fraction& x)
    { // 若分数 * this 大于分数 x 的值则返回 1, 否则返回 0
        if (nume * x.deno > deno * x.nume)
```

```

        return 1;
    else
        return 0;
    }
    Fraction& operator ++ ();
    // 函数声明 具体函数定义如 8.1 节给出的那样
    Fraction operator ++ (int);
    // 函数声明 具体函数定义如 8.1 节给出的那样
}; // 分数类定义结束

istream& operator >> (istream& istr, Fraction& x)
// 从键盘上按规定格式输入一个分数到 x 中
{
    char ch; // 用 ch 保存分数输入中的除号
    cout << "Input a fraction:";
    istr >> x.nume >> ch >> x.deno;
    if(x.deno == 0) {
        cerr << "除数为 0!" << endl;
        exit(1);
    }
    return istr;
}

ostream& operator << (ostream& ostr, Fraction& x)
// 按规定格式输出 x 中的分数
{
    ostr << x.nume << '/' << x.deno << endl;
    return ostr;
}

void main()
{
    Fraction a(5,8),b,c; // 定义 3 个类对象,前一个将自动调用
    // 带参构造函数,后两个将自动调用无参构造函数
    cin >> b; // 调用提取操作符重载函数,给分数 b 输入新值
    b++; // 分数 b 后加 1
    c = a+b; // a+b 赋给 c
    cout << a; // 输出分数 a
    cout << b; // 输出分数 b
    cout << c; // 输出分数 c
    cout << (a > b ? 1:0) << endl; // a 大于 b 则输出 1 否则输出 0
}

```

该程序运行结果为:

```

Input a fraction:3/7
5/8
10/7
115/56
0

```

声明友元函数或友元类的语句以关键字 `friend` 开始,后跟一个函数或类的声明。此语



句可以放在类中的任何位置,与它前面使用的任一访问权限关键字无关。

该程序的分数类中说明了两个友元函数,一个为用于从键盘上输入分数的提取操作符重载函数,另一个为用于向屏幕输出分数的插入操作符重载函数。在这两个友元函数中都直接存取了分数对象  $x$  中的私有数据成员分子和分母的值。若不把它们规定为分数类的友元,则是不允许它们访问该类的私有成员的。下面给出的输入输出函数是不把它们规定为分数类的友元的情况,它们只能通过调用分数类的成员函数访问数据成员,从而大大降低了访问分数类对象的速度。

```
istream& operator >> (istream& istr, Frac& x)
{
    char ch; // 用 ch 保存分数输入中的除号
    cout << "Input a fraction:";
    int n, d;
    istr >> n >> ch >> d;
    x.InNum(n); x.InDen(d);
    if(d == 0) {
        cerr << "除数为 0!" << endl;
        exit(1);
    }
    return istr;
}

ostream& operator << (ostream& ostr, Frac& x)
{
    ostr << x.GetNum() << "/" << x.GetDen() << endl;
    return ostr;
}
```

用于分数输入或输出的操作符重载函数,其第一个操作数为输入流或输出流对象,第二个操作数为类对象,所以不能把它们改写为成员函数,只能作为普通函数定义和使用。当一个普通函数的第一个参数为类对象时,都可以改写为该类的成员函数,否则不能改写。改写得到的成员函数省略了原来的第一个参数,它由 `this` 指针所指向的对象所代替。

例 2. 下面的程序定义了两个类,一个为 `Point` 类,另一个为 `Circle` 类,其中后者被声明为前者的友元类,这样在后者的所有成员函数中都可以访问前者类对象中的私有成员。

```
#include <iostream.h>
class Point { // 坐标点类
    int x, y; // 点的横坐标和纵坐标
    friend class Circle;
    // 声明 Circle 类为 Point 类的友元类
public:
    Point() { // 无参构造函数,置坐标点为原点(0,0)
        x = 0; y = 0;
    }
    Point(int xx, int yy) { // 带参构造函数,置坐标点为形参的值
        x = xx; y = yy;
    }
};
```

```

class Circle { // 圆类
    Point centre; // 圆心坐标
    int radius; // 圆的半径
public:
    Circle() { // 无参构造函数,置圆心为(0,0),置半径为0.
        radius = 0;
    }
    Circle(int a, int b, int r): centre(a,b)
    { // 带参构造函数,置圆心为(a,b),置半径为r
        radius = r;
    }
    float Area() { // 计算圆的面积
        return float(radius * radius * 3.14159);
    }
    void Print() { // 输出圆心坐标和半径
        cout << '(' << centre.x << ', ' << centre.y;
        cout << ')' << radius << endl;
    }
    void Modify(int xm=0, int ym=0, int rm=0)
    { // 修改圆心坐标和半径,使之增加相应的值
        centre.x += xm;
        centre.y += ym;
        radius += rm;
    }
};

void main()
{
    Circle c1,c2(2,3,4);
    cout << "c1: "; c1.Print();
    cout << "c2: "; c2.Print();
    c1.Modify(2,5);
    cout << "Modify c1: "; c1.Print();
    cout << c1.Area() << ' ' << c2.Area() << endl;
}

```

在一个类中,数据成员可以为除本身类型之外(但本身类型可作为指针类型使用)的任何类型,当然也包括类类型在内。当数据成员为一个类类型时,则对该数据成员的初始化只能通过初始化表实现,不能通过函数体实现。在初始化表中通过使用“类数据成员(实参表)”的格式调用相应的构造函数实现对该数据成员的初始化。若没有在初始化表中对类数据成员初始化,则系统隐含调用该类的无参构造函数使之初始化,相当于在初始化表中使用了“类数据成员()”调用项。若类中没有定义任何构造函数,则将使用系统隐含定义无参构造函数,若类中定义了构造函数,但没有定义无参构造函数,则对类数据成员调用无参构造函数初始化时将出现编译错误。

在上面程序的 Circle 类中,无参构造函数首先调用 Point 类的无参构造函数给数据成员 centre 中的 x 和 y 分别赋初值 0,接着执行函数体给数据成员 radius 赋初值 0。带参构造函数首先调用 Point 类的带参构造函数给数据成员 centre 中的 x 和 y 分别赋初值为参数 a 和 b 的值,接着执行函数体给数据成员 radius 赋初值为参数 r 的值。

由于 Circle 类是 Point 类的友元类,所以在 Circle 类中的任何成员函数都可以使用 Point 类中的私有成员。Circle 类包括有五个成员函数,其中 Print 成员函数和 Modify 成员函数使用了 Point 类中的私有数据成员。若没有把 Circle 类定义为 Point 类的友元,则这种使用是不允许的,在程序编译时将出现语法错误。

在上面程序的主函数中,第一条语句定义了两个圆类对象 c1 和 c2,c1 调用了无参构造函数,c2 调用了带参构造函数;第二条和第三条语句显示出圆 c1 和 c2 中的数据;第四条语句修改 c1 中的数据,使圆心 x 坐标和 y 坐标的值分别增加 2 和 5,使半径 radius 的值增加默认值 0;第五条语句显示出修改后的 c1 中的数据,第六条语句显示出 c1 和 c2 的面积。该程序的运行结果如下:

```
c1: (0,0)0
c2: (2,3)4
Modify c1: (2,5)0
0 50.2654
```

例 3. 下面程序包含有三个类,其中前两个为例 2 中的类,第三个为 Group 类。Group 类需要使用前两个类中的私有成员,所以在前两个类中分别声明 Group 类为它们的友元。

```
#include<iostream.h>
class Point { // 坐标点类
    int x,y; // 点的横坐标和纵坐标
    friend class Circle;
    // 声明 Circle 类为 Point 类的友元类
    friend class Group;
    // 声明 Group 类为 Point 类的友元类
public:
    // 无参和带参构造函数与上例相同
};

class Circle { // 圆类
    Point centre; // 圆心坐标
    int radius; // 圆的半径
    friend class Group;
    // 声明 Group 类为 Circle 类的友元类
public:
    // 所有公用函数与上例相同
};

class Group { // 圆组类
    Circle* a; // 圆类数组指针
    int n; // 圆类数组长度
public:
    Group(): a(0), n(0) {} // 无参构造函数
    Group(int aa[][3], int nn) { // 带参构造函数
        n=nn;
        a=new Circle[n];
        // 每个元素均调用 Circle 类的无参构造函数进行初始化
        for(int i=0; i<n; i++) { // 向每个元素输入圆的数据
            a[i].centre.x=aa[i][0];
```

```

        a[i].centre.y = aa[i][1];
        a[i].radius = aa[i][2];
    }
}

~Group() { // 析构函数
    delete []a;
    cout << "释放 Group 类对象中的动态存储空间!" << endl;
}

void OutputMax() { // 输出类对象中保存的最大圆的数据
    int k = 0;
    for(int i = 1; i < n; i++)
        if(a[i].radius > a[k].radius) k = i;
    cout << "(" << a[k].centre.x << ", ";
    cout << a[k].centre.y << ")" << a[k].radius;
    cout << ": " << a[k].Area() << endl;
}

};

void main()
{
    int g[][3] = {{2,3,4},{5,1,7},{4,4,5},{2,4,8},{1,3,6}};
    Group a(g,5);
    a.OutputMax();
}

```

在 Group 类中包含有 Circle 类指针成员 a 和整数成员 n, 用 a 指向 Circle 类类型的动态分配的数组空间, 用 n 表示该数组的长度。该类中包含有两个构造函数, 一个为无参构造函数, 它置 a 和 n 的初值均为 0, 指针值为 0 表示一个空(NULL)指针; 第二个为带参构造函数, 它首先为数据成员 n 赋初值, 接着动态分配具有 n 个 Circle 类元素的存储空间, 并由指针成员 a 所指向, 对于每个元素都将自动调用该类的无参构造函数进行初始化, 若没有无参构造函数可供调用, 则无法生成类数组, 包括动态和非动态定义的数组, 然后把形参数组 aa[nn][3] 中的每一行的 3 个整数分别赋给 a 数组的相应元素中, 使它们分别成为该元素(即一个圆对象)的圆心横坐标、圆心纵坐标和圆的半径。Group 类中定义有析构函数, 它释放为类对象动态分配的数组空间。在 Group 类中除了定义构造函数和析构函数外, 只定义了一个成员函数 OutputMax, 其作用是输出数据 a 中保存的最大圆的圆心坐标、半径和面积。该程序运行结果如下:

```

(2,4)8: 201.062
释放 Group 类对象中的动态存储空间!

```

## 习题八

### (一) 单选题

1. 当类中一个字符指针成员指向具有 n 个字节的存储空间时, 它所能存储字符串的最大长度为\_\_\_\_\_。

• 256 •

- A n                      B n+1                      C n-1                      D n-2
2. 假定 AB 为一个类,则该类的拷贝构造函数的声明语句为\_\_\_\_\_。
- A AB&(AB x);    B AB(AB x)                      C AB(AB&);                      D AB(AB \* x)
3. 对类对象成员的初始化是通过执行构造函数中的\_\_\_\_\_完成的。
- A 初始化表                      B 函数体                      C 参数表                      D 基类表
4. 假定 AB 为一个类,则执行“AB a, b(3), \* p;”语句时,自动调用该类构造函数的次数为\_\_\_\_\_。
- A 2                      B 3                      C 4                      D 5
5. 假定 AB 为一个类,则执行“AB a(4), b[3], \* p[2];”语句时,自动调用该类构造函数的次数为\_\_\_\_\_。
- A 3                      B 4                      C 6                      D 9
6. 假定 AB 为一个类,px 为指向该类动态对象数组的指针,该数组长度为 n,则执行“delete []px;”语句时,自动调用该类析构函数的次数为\_\_\_\_\_。
- A 0                      B 1                      C n                      D n+1
7. 对于类中定义的成员,其隐含访问权限为\_\_\_\_\_。
- A public                      B protected                      C private                      D static
8. 对于结构中定义的成员,其隐含访问权限为\_\_\_\_\_。
- A public                      B protected                      C private                      D static
9. 为了使类中的成员不能被类外的函数通过成员操作符访问,则不应把该成员的访问权限定义为\_\_\_\_\_。
- A public                      B protected                      C private                      D static
10. 一个类的友元函数或友元类能够通过成员操作符访问该类的\_\_\_\_\_。
- A 私有成员                      B 保护成员                      C 公用成员                      D 所有成员
11. 假定要对类 AB 定义加号操作符重载成员函数,实现两个 AB 类对象的加法,并返回相加结果,则该成员函数的声明语句为:\_\_\_\_\_
- A AB operator + (AB& a, AB& b);                      B AB operator + (AB& a);
- C operator + (AB a);                      D AB& operator + ();

## (二) 填空题

1. 在定义类对象的语句执行时,系统在建立每个对象的过程中将自动调用该类的\_\_\_\_\_使其初始化。
2. 当一个类对象被撤消时将自动调用该类的\_\_\_\_\_。
3. 对一个类中的数据成员的初始化可以通过构造函数中的\_\_\_\_\_实现,也可以通过构造函数中的\_\_\_\_\_实现。
4. 设 px 是指向一个类动态对象的指针变量,则执行“delete px;”语句时,将自动调用该类的\_\_\_\_\_。
5. 当一个类对象离开它的作用域时,系统将自动调用该类的\_\_\_\_\_。
6. 假定一个类对象数组为 A[N],当离开它的作用域时,系统自动调用该类析构函数的次数为\_\_\_\_\_。

7. 假定 AB 为一个类,则执行“AB a[10];”语句时,系统自动调用该类构造函数的次数为\_\_\_\_\_。
8. 假定用户没有给一个名为 AB 的类定义构造函数,则系统为其隐含定义的构造函数为\_\_\_\_\_。
9. 假定用户没有给一个名为 AB 的类定义析构函数,则系统为其隐含定义的析构函数为\_\_\_\_\_。
10. 若需要把一个函数“void F();”定义为一个类 AB 的友元函数,则应在类 AB 的定义中加入一条语句:\_\_\_\_\_。
11. 若需要把一个类 AB 定义为一个类 CD 的友元类,则应在类 CD 的定义中加入一条语句:\_\_\_\_\_。

### (三) 写出下列每个程序运行后的输出结果

1. 

```
#include <iostream.h>
class A {
    int a,b;
public:
    A() {a=b=0;}
    A(int aa, int bb) {
        a=aa; b=bb;
        cout << a << ' ' << b << endl;
    }
};
void main() {
    A x,y(2,3),z(4,5);
}
```
2. 

```
#include <iostream.h>
class A {
    int a,b;
public:
    A(int aa=0, int bb=0): a(aa), b(bb) {
        cout << "Constructor!" << a+b << endl;
    }
};
void main() {
    A x,y(2,3),z(y);
}
```
3. 

```
#include <iostream.h>
class A {
    int * a;
public:
    A(int aa=0) {
        a=new int(aa);
        cout << "Constructor!" << *a << endl;
    }
};
```

```

void main() {
    A x[2];
    A *p = new A(5);
    delete p;
}

4. #include<iostream.h>
class A {
    int a;
public:
    A(int aa = 0): a(aa) {}
    ~A() {cout << "Destructor!" << a << endl;}
};
void main() {
    A x(5);
    A *p = new A(10);
    delete p;
}

5. #include<iostream.h>
class A {
    int * a;
public:
    A(int x) {
        a = new int(x);
        cout << "Constructor!" << *a << endl;
    }
    ~A() {delete a; cout << "Destructor!" << endl;}
};
void main() {
    A x(3), *p;
    p = new A(5);
    delete p;
}

6. #include<iostream.h>
#include<stdlib.h>
class A {
    int a,b; char op;
public:
    A(int aa, int bb, char ch) {a = aa; b = bb; op = ch;}
    int Comp() {
        switch(op) {
            case '+': return a + b;
            case '-': return a - b;
            case '*': return a * b;
            case '/': if(b != 0) return a/b; else exit(1);
            case '%': if(b != 0) return a % b; else exit(1);
            default: exit(1);
        }
    }
}

```

```

        void SetA(int aa, int bb, char ch) {
            a = aa; b = bb; op = ch;
        }
    };

    void main(void) {
        A x(3,5,'*');
        int a = x.Comp();
        x.SetA(4,9,'+');
        a += x.Comp();
        x.SetA(8,5,'%');
        a += x.Comp();
        cout << "a = " << a << endl;
    }

```

```

7. #include <iostream.h>
class B {
    int a,b;
public:
    B() {a=b=0;}
    B(int aa, int bb) {a=aa; b=bb;}
    B operator + (B& x) {
        B r;
        r.a = a + x.a;
        r.b = b + x.b;
        return r;
    }
    B operator - (B& x) {
        B r;
        r.a = a - x.a;
        r.b = b - x.b;
        return r;
    }
    void OutB() {
        cout << a << ' ' << b << endl;
    }
};

void main() {
    B x(3,5), y(8,4), z1,z2;
    z1 = x + y; z2 = x - y;
    z1.OutB(); z2.OutB();
}

```

```

8. #include <iostream.h>
   #include <stdlib.h>
class RMB {
    int yuan,jiao,fen;
    void Norm() {
        if(fen > 9) {
            jiao += fen/10; fen %= 10;
        }
        if(jiao > 9) {

```



```

        yuan += jiao/10; jiao %= 10;
    }
}

void Error() {
    cout << "data not negative!" << endl;
    exit(1);
}

public:
RMB(int a=0, int b=0, int c=0) {
    if(a<0 || b<0 || c<0) Error();
    yuan=a; jiao=b; fen=c;
    Norm();
}

void SetValue(int a=0, int b=0, int c=0) {
    if(a<0 || b<0 || c<0) Error();
    yuan=a; jiao=b; fen=c;
    Norm();
}

void Output() {
    cout << yuan << " yuan, " << jiao << " jiao, " << fen << " fen" << endl;
}

RMB& operator + (RMB& r) {
    yuan += r.yuan; jiao += r.jiao; fen += r.fen;
    Norm();
    return *this;
}

RMB& operator - (RMB& r) {
    fen = fen + jiao * 10 + yuan * 100;
    jiao = yuan = 0;
    if(fen >= r.fen)
        fen -= r.fen;
    else
        Error();
    Norm();
    jiao = jiao + yuan * 10;
    yuan = 0;
    if(jiao >= r.jiao)
        jiao -= r.jiao;
    else
        Error();
    Norm();
    if(yuan >= r.yuan)
        yuan -= r.yuan;
    else
        Error();
    Norm();
    return *this;
}

RMB& operator * (int n) {
    if(n<0) Error();
    fen *= n; jiao *= n; yuan *= n;
}

```

```

        Norm();
        return *this;
    }
};

void main()
{
    RMB a,b(4,5,9),c,d(b),e;
    a.SetVolume(2,8,5);
    c+a; c+b;
    d-a; e=d; e*3;
    a.Output();b.Output();c.Output();d.Output();e.Output();
}

```

```

9. #include<iostream.h>
   #include<string.h>
   class Strings {
       char* ps;
       friend ostream& operator << (ostream& ostr, Strings& s);
   public:
       Strings(): ps(0) {}
       Strings(char* ss) {
           ps = new char[strlen(ss) + 1];
           strcpy(ps,ss);
       }
       Strings(Strings& s) {
           ps = new char[strlen(s.ps) + 1];
           strcpy(ps,s.ps);
       }
       Strings& operator = (Strings& s) {
           delete []ps;
           ps = new char[strlen(s.ps) + 1];
           strcpy(ps,s.ps);
           return *this;
       }
       ~Strings() {
           delete []ps;
           cout <<"release dynamic memory space!"<< endl;
       }
       void Setps(char* ss) {
           delete []ps;
           ps = new char[strlen(ss) + 1];
           strcpy(ps,ss);
       }
       int operator > (Strings& s) {
           if(strcmp(ps,s.ps) > 0)
               return 1;
           else
               return 0;
       }
       int operator == (Strings& s) {

```

```

        if(strcmp(ps,s.ps) == 0)
            return 1;
        else
            return 0;
    }
    int operator< (Strings& s) {
        if(strcmp(ps,s.ps) < 0)
            return 1;
        else
            return 0;
    }
    Strings operator+ (Strings& s) {
        Strings r;
        r.ps = new char[strlen(this->ps) + strlen(s.ps) + 1];
        strcpy(r.ps, this->ps);
        strcat(r.ps, s.ps);
        return r;
    }
};

ostream& operator << (ostream& ostr, Strings& s) {
    if(s.ps) ostr << s.ps << ' ';
    return ostr;
}

void main()
{
    Strings s1,s2("C++ language"),s3;
    s3.Setps("programing.");
    s1 = s2 + s3;
    cout << s1 << endl;
    cout << s2 << endl;
    cout << s3 << endl;
    if(s1 > s2) cout << "s1 > s2" << endl;
    else if(s1 == s2) cout << "s1 == s2" << endl;
    else cout << "s1 < s2" << endl;
}

```

#### (四) 按题目要求编写出程序、函数或类

1. 下面是定义二次多项式  $ax^2 + bx + c$  所对应的类

```

#include <iostream.h>
#include <math.h>
class Quadratic { // 二次多项式类
double a,b,c;
public:
    Quadratic() {a=b=c=0;}
    Quadratic(double aa, double bb, double cc);
    Quadratic operator + (Quadratic& x);
    Quadratic operator - (Quadratic& x);

```

```

double Compute(double x);
int Root(double& r1, double& r2);
void Print();
};

```

其中加、减操作符重载函数完成 \* this 和 x 的加或减运算,并将运算结果返回;Compute 函数根据 x 的值计算二次多项式  $ax^2 + bx + c$  的值并返回;Root 函数求出二次方程  $ax^2 + bx + c = 0$  的根,要求当不是二次方程(即  $a=0$ )时返回 -1,当有实根时返回 1,并由引用参数 r1 和 r2 带回这两个实根,当无实根时返回 0;Print 函数按  $ax^2 + bx + c$  的格式( $x^2$  用  $x * * 2$  表示)输出二次多项式,并且当 b 和 c 的值为负时,其前面不能出现加号。试写出在类定义中声明的每个成员函数在体外的定义。

2. 请定义一个矩形类(Rectangle),私有数据成员为矩形的长度(len)和宽度(wid),无参构造函数置 len 和 wid 为 0,带参构造函数置 len 和 wid 为对应形参的值,另外还包括求矩形周长、求矩形面积、取矩形长度、取矩形宽度、修改矩形长度和宽度为对应形参的值、输出矩形尺寸等公用成员函数。要求输出矩形尺寸的格式为“length: 长度, width: 宽度”。

## 第九章 类的继承与多态性

### 9.1 类的继承

类是一种抽象数据类型,是对具有共同属性和行为的对象(事物)的抽象描述。但通常为了处理问题的方便,对事物按层进行分解,使得处于顶层(上层)的抽象事物具有处于底层(下层)抽象事物的共同特征,而处于底层的抽象事物除了具有顶层抽象事物的所有特征外,还具有本身所专有的特征。例如对于建筑物来说,它有施工单位、竣工日期等特征;而建筑物又可细分为房屋、桥梁和纪念塔等三类,它们除了具有建筑物的共同特征外,还各自具有自己的特征,如房屋有建筑面积,桥梁有建筑高度、宽度和长度,纪念塔有塔高和形状等特征;房屋又可细分为平房和楼房两类,平房和楼房除了具有房屋的共同特征外,还具有自己的特征,如平房有庭院面积,楼房有楼层数和电梯数等特征;楼房又可细分为办公楼和居民楼两类,它们除了具有楼房的公共特征外,办公楼还具有值班电话,居民楼还具有居民户数和居住人数等特征。可用图 9-1 表示它们之间的层次关系。

在C++中允许定义类之间的继承关系。当一个类继承另一个类时,这个类被称为继承类、派生类或子类,另一个类被称为被继承类、基类或父类。子类能够继承父类的全部特征,包括所有的数据成员和成员函数,并且子类还能够定义父类所没有的、属于自己的特征,即自己的数据成员和成员函数。通过类的继承关系,使得一些类的代码可以为定义另一些类所重用,避免了代码的重新书写和调试,能够开发出便于维护和扩充、可靠性高的软件。所以,类的继承是软件开发中的一项重要技术。



图 9-1 建筑物类层次图

#### 9.1.1 派生类定义的格式

```
class <派生类名> : <基类表> { <成员表> };
```

它同一般类的定义格式大体相同,只是在类名和左花括号之间增添了一个冒号和一个基类表。

语句定义中的派生类名是新定义的类型标识符,它是基类表中所给基类的一个派生类;基类表中包含有一个或多个用逗号分开的类项,每个类项为一个已被定义的作为基类使用的类名,它前面可以带有继承权限指明符用以规定被继承的权限;花括号内的成员表是为该派生类定义的数据成员和成员函数的列表。

基类表中每个基类名前面可以使用的继承权限指明符仍为类成员表中为规定成员访问权限所使用的指明符 public、private 或 protected,它们分别表示派生类公用(公有)、私有或保

护继承该基类。若一个基类名前没有使用任一指明符也是允许的,对于 class 定义语句来说,隐含为 private 指明符,即派生类私有继承该基类,对于 struct 定义语句来说,隐含为 public 指明符,即派生类公用继承该基类,这种规定与定义它们的成员时默认的访问权限的规定完全相同。

当一个基类被派生类公用继承时,则基类中的所有 public 成员也同时成为派生类中的 public 成员,基类中的所有 protected 成员也同时成为派生类中的 protected 成员,基类中的所有 private 成员不转换为派生类中的任何成员,仍作为基类的私有成员保留在基类中,也可以说同时保留在派生类中,因为派生类继承了基类中的所有成员。由于基类的私有成员没有同时成为派生类中的成员,所以派生类的成员函数无法直接访问它们,只能通过基类提供的公用或保护成员函数来间接访问。当然,若把派生类定义为基类的友元,则可直接访问其私有成员。

当一个基类被派生类私有继承时,则基类中的所有 public 成员和所有 protected 成员将同时成为派生类中的 private 成员,基类中的所有 private 成员仍只作为基类的私有成员存在,不转换为派生类中的任何成员。

当一个基类被派生类保护继承时,则基类中的所有 public 成员和所有 protected 成员将同时成为派生类中的 protected 成员,基类中的所有 private 成员同上述两种继承一样,仍只能作为基类的私有成员存在,不是派生类的成员。

无论任何一个类,无论它的成员是靠继承而来的,还是自己定义的,都属于自己的成员,该类的成员函数能够访问该类中具有任何访问权限的成员,同时也能够访问其他类中具有公用访问权限的成员和类外的对象与函数,不能访问其他类中的保护成员和私有成员,即使其他类是自己继承的类,或自己成员所属的类也是如此。

在一个派生类中,其成员由两部分组成,一部分是从基类继承得到的,另一部分是自己定义的新成员,所有这些成员也分为公用(public)、私有(private)和保护(protected)这三种访问属性。另外,从基类继承下来的全部成员构成派生类的基类部分,此部分的私有成员是派生类不能直接访问的,其公用和保护成员是派生类可以直接访问的,因为它们已同时成为了派生类中的成员,但在派生类中的访问属性可能有改变,视对基类的继承权限而定。带有一个基类的派生类的构成如图 9-2 所示。

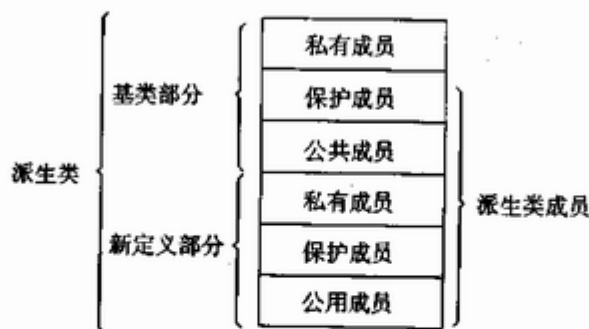


图 9-2 派生类构成示意图

当派生类公用继承基类时,派生类中的公用成员包括基类部分的公用成员和新定义部

分的公用成员,保护成员包括基类部分的保护成员和新定义部分的保护成员,私有成员仅为新定义部分的私有成员。当派生类保护继承基类时,派生类中的公用成员仅为新定义部分的公用成员,保护成员包括基类部分的公用成员和保护成员以及新定义部分的保护成员,私有成员仅为新定义部分的私有成员。当派生类私有继承基类时,派生类中的公用成员仅为新定义部分的公用成员,保护成员也仅为新定义部分的保护成员,私有成员包括基类部分的公用成员和保护成员以及新定义部分的私有成员。

每个派生类对象所占有的存储空间的大小等于其基类部分的所有数据成员占有的存储空间的大小与新定义部分的所有数据成员占有的存储空间大小的总和,并且前面的存储空间分配给基类部分的数据成员使用,后面的存储空间分配给新定义部分的数据成员使用。

### 9.1.2 格式举例

```
1. struct X {
    int a;
};
struct Y: X {
    int b;
    int c;
};
```

这里 X 和 Y 均为结构,所以它们的成员默认为公用访问属性,Y 对 X 的继承也默认为公用继承。派生类 Y 中包含有三个公用成员 a、b 和 c,其中 a 又同时为基类 X 部分的公用成员。由于 Y 中的成员都是公用的,所以外部函数可以直接访问以 Y 为类型的对象中的任何成员。每个 Y 类型的对象具有 12 个字节的存储空间,其中前 4 个字节存储数据成员 a,接着 4 个字节存储数据成员 b,最后 4 个字节存储数据成员 c。假定 y 是 Y 结构类型的一个对象,则 y.a、y.b 和 y.c 都是有效的表示,利用它们可以访问 y 中的指定成员。当然 y.a 也可以表示为 y.X::a,因为 a 同时又是 y 结构中的基结构类型 X 中的成员。

```
2. class A {
    int a1;
protected:
    int a2;
public:
    int a3;
    A() {a1 = a2 = a3 = 0;}
    A(int x1, int x2, int x3):a1(x1),a2(x2),a3(x3) {}
    void OutA() {cout << a1 << ' ' << a2 << ' ' << a3 << endl;}
    int Geta1() {return a1;}
};

class B: public A {
    int b1;
protected:
    int b2;
public:
    int b3;
```

```

    B() {b1 = b2 = b3 = 0;}
    B(int x1, int x2, int x3): A(x1,x2,x3) {
        b1 = x1 + 1; b2 = x2 + 2; b3 = x3 + 3;
    }
    void OutB() {
        A::OutA();
        cout << b1 << " " << b2 << " " << b3 << endl;
    }
    int Sum() {
        return Geta1() * b1 + a2 * b2 + a3 * b3;
    }
};

void main()
{
    B b1,b2(1,2,3);
    b1.OutB();
    b2.OutB();
    cout << b1.Sum() << " " << b2.Sum() << endl;
    cout << sizeof(B) << endl;
};

```

在这个例子中,定义了两个类 A 和 B,并且把 B 定义为 A 的派生类,则 A 是 B 的基类,B 公用继承了 A 中的所有成员。在类 A 中定义有三个整数成员 a1,a2 和 a3,它们的访问权限依次为私有、保护和公用;定义有一个无参构造函数,它对所有数据成员赋初值为 0;定义有一个带参构造函数,它对三个数据成员分别赋初值为形参 x1,x2 和 x3 的值;定义有一个 OutA 公用成员函数,用于输出所有数据成员的值;定义有一个 Geta 公用成员函数,用以返回私有成员 a1 的值。

在类 B 中除了继承类 A 中的所有成员外,还定义有三个整数成员 b1,b2 和 b3,它们的访问权限也相应为私有、保护和公用,以及定义有四个公用成员函数。第一个为无参构造函数,执行时首先隐含调用基类 A 的无参构造函数给属于基类 A 中的数据成员初始化,接着执行函数体给新定义的数据成员初始化为 0。第二个为带参构造函数,它首先利用初值项 A(x1,x2,x3)调用基类 A 的带参构造函数,对属于 A 中的数据成员初始化,接着执行函数体对新定义的数据成员初始化。第三个为 OutB 函数,它首先调用 OutA()函数输出基类 A 中数据成员的值,然后输出 B 类中新定义的所有数据成员的值。由于 OutA()函数既是基类 A 部分的公用函数,又是 B 类继承过来的公用函数,所以写成 OutA()或 A::OutA()的调用形式完全相同。在派生类的成员函数中,使用基类名和类区分符作为成员名的前缀能够访问既属于基类又属于派生类的成员。第四个为 Sum 函数,它返回 a1 \* b1 + a2 \* b2 + a3 \* b3 的值,由于 a1 是基类 A 的私有成员,类 A 外的函数无法访问,所以只能由类 A 提供的成员函数 Geta1()得到它的值,因此在返回表达式中不能直接书写为 a1,而应书写为 Geta1()。

在派生类的构造函数中,对属于基类的成员进行初始化是通过在初值表中给出具有“基类名(实参表)”格式的初值项调用基类的构造函数来实现的。若初值表中没有给出调用基类构造函数的初值项,则自动调用基类的无参构造函数进行初始化,就如同在初值表中使用了“基类名()”初值项进行调用一样。



在一个派生类中同时含有类成员时,则类成员的初始化和基类成员的初始化一样,都必须在构造函数的初值表中给出初值项,以此调用相应的构造函数来实现,当省略初值项时则调用相应的无参构造函数来实现。派生类构造函数的执行顺序是:首先调用基类构造函数实现对基类成员的初始化,接着调用成员所属类的构造函数实现类成员的初始化,最后实现对新定义的非类成员的初始化,最后一步的初始化可以通过初值表,也可以通过函数体进行。

此例中主函数定义了派生类 B 的两个对象 b1 和 b2, b1 对象通过调用无参构造函数进行初始化,使得所有数据成员的值均为 0, b2 对象通过调用带参构造函数进行初始化,使得 b2 中属于基类 A 部分的数据成员 a1, a2 和 a3 被初始化为 1, 2 和 3, 属于新定义部分的数据成员 b1, b2 和 b3 被初始化为 2, 4 和 6, 当然属于基类 A 部分的 a2 和 a3 也同时为 b2 的保护成员和公用成员。该程序的运行结果如下:

```

0 0 0
0 0 0
1 2 3
2 4 6
0 28
24

3. class AA {
    protected:
        int a;
    public:
        AA(int x=0): a(x) {}
        int Geta() {
            return a;
        }
};

class BB {
    protected:
        int b;
    public:
        BB(int x=0): b(x) {}
};

class CC: public AA, public BB {
    int c;
    AA d;
    public:
        CC() {c=0;}
        CC(int x, int y, int z): AA(x), BB(y),
            c(z), d(x+y+z)
        {}
        void OutCC() {
            cout << "a=" << a << ", b=" << b;
            cout << ", c=" << c << endl;
            cout << "d.a=" << d.Geta() << endl;
        }
};

```

```

void main()
{
    CC c1,c2(3,4,5);
    c1.OutCC();
    c2.OutCC();
    cout << sizeof(CC) << endl;
}

```

在这个例子中定义了三个类,其中 AA 和 BB 均为 CC 的基类,派生类 CC 既继承了 AA 的全部成员又继承了 BB 的全部成员,称 CC 为多继承的派生类,相反当只有一个基类时则称为单继承。在类 AA 和 BB 的构造函数中,其参数都带有默认值,所以它们既可以作为无参构造函数接受无参调用,又可以作为带参构造函数接受带参调用。在执行类 CC 的无参或带参构造函数时,无论初值表中各初值项的前后次序如何,都是首先调用类 AA 的构造函数对 CC 类中属于 AA 类的成员初始化,接着调用类 BB 的构造函数,对 CC 类中属于 BB 类的成员初始化,再接着调用类 AA 的构造函数对类数据成员 d 进行初始化,最后初始化数据成员 c。

在执行类 CC 中的成员函数 OutCC()时,输出保护数据成员 a 和 b 的值,私有数据成员 c 的值,以及私有数据成员 d 中的 a 成员的值。由于 d.a 是 d 所属类 AA 的保护成员,类 AA 外不能够直接访问,所以若写成 d.a 进行访问是错误的,只能采用类 AA 提供的公共成员函数 Geta()读取它的值。

在主函数中定义了类 CC 的两个对象 c1 和 c2。当 c1 自动调用类 CC 的无参构造函数进行初始化时,首先调用 AA 的无参构造函数对 c1.a 初始化为 0,接着调用 BB 的无参构造函数对 c1.b 初始化为 0,再接着调用 AA 类的无参构造函数对 c1.d.a 初始化为 0,最后执行函数体对 c1.c 初始化为 0。当 c2 自动调用类 CC 的带参构造函数时,首先把实参值 3,4 和 5 分别赋给形参 x,y 和 z,接着依次通过初值表中给出的初始项 AA(x),BB(y)和 d(x+y+z)调用对应类的构造函数,分别给 c2.a,c2.b 和 c2.d.a 赋初值为 3,4 和 12,最后执行初值项 c(z)的操作给 c2.c 赋初值为 5。该程序运行结果为:

```

a=0, b=0, c=0
d.a=0
a=3, b=4, c=5
d.a=12
16

```

```

4. class A1 {
    protected:
        int a;
    public:
        A1(int x=0) { a=x;}
        int Square() {return a*a;}
};
class A2: public A1 {
    public:
        int b;
        A2(int aa=0, int bb=0): A1(aa) {

```

```

        b=bb;
    }
    int Square() { return b*b;}
};

class A3: public A2 {
    char a;
    int b;
    int c;
public:
    A3() {a='\0'; b=0; c=0;}
    A3(char ch, int x): A2(x, 3*x) {
        a=ch;
        b=2*x;
        c=4*x;
    }
    int Square() {return (b+c)*(b+c);}
    void OutA3() {
        cout <<"a="<<'\' "<<a <<'\' "<<" b="<<b;
        cout <<" c="<<c <<endl;
        cout <<"A2::b="<<A2::b <<endl;
        cout <<"A1::a="<<A1::a <<endl; // 也可写为 A2::a
        cout <<Square() <<'\' "<<A2::Square() <<'\' ";
        cout <<A1::Square() <<endl;
    }
};

void main()
{
    A3 d1,d2('d', 5);
    cout <<d1.Square() <<'\' "<<d2.Square() <<endl;
    cout <<d2.A2::Square() <<'\' "<<d2.A1::Square() <<endl;
    cout <<sizeof(d2) <<endl <<endl;
    d1.OutA3();
    cout <<endl;
    d2.OutA3();
}

```

在这个例子中, A2 公用继承了 A1, A3 又公用继承了 A2, 所以 A3 包含有在这三个类中定义的所有成员。在派生类中定义的成员可以与基类或基类的基类中定义的成员具有相同或不同的名字, 若不同时, 成员名能够惟一确定所处的位置, 不必加类名和类区分符作为前缀限定; 若相同时, 对于不加类名和类区分符作为前缀的成员名, 则表示当前类中定义的成员, 若要访问与派生类同名的基类中的成员, 则必须在成员名前加上基类名和类区分符。如在三个类 A1, A2 和 A3 中都定义有成员函数 Square, 并且都是它们的公用成员, 由于继承关系, 它们都是 A3 中的公用成员。在 A3 的成员函数中使用 Square() 表示调用该类中定义的成员函数, 使用 A2::Square() 表示调用基类 A2 中定义的成员函数, 使用 A1::Square() 表示调用间接基类 A1 (它是 A2 的直接基类, 是 A3 的间接基类, 又称传递基类) 中定义的成员函数。又如在 A3 和间接基类 A1 中都定义有数据成员 a, 并且分别是各自的私有成员和保护成员, 由于继承关系, A1 中定义的 a 同时成为 A3 中的保护成员, 允许在 A3 的成员函数中使

用,当成员名 a 前不加基类名和类区分符时则表示访问在当前类即 A3 中定义的私有成员 a,当成员名 a 前加上基类名 A1 和类区分符时则表示访问在基类 A1 中定义的保护成员 a。

在类 A3 中包含有无参和带参两个构造函数,当定义的 A3 类的对象调用任一构造函数执行时,都首先调用基类 A2 的构造函数,此时又首先调用 A2 的基类 A1 的构造函数,A1 的构造函数执行结束后返回到 A2 的构造函数执行,待 A2 的构造函数执行结束后,又返回到 A3 的构造函数执行。所以执行一个类的构造函数时,最深(即最顶)层的基类构造函数最先执行,接着执行次深层的基类的构造函数,依次向下,最后执行当前类的构造函数对新定义的成员初始化。当然若不含有任何基类,则直接执行当前类的构造函数。

此程序的输出结果如下,请读者结合主函数中的语句自行分析输出结果。

```
0 361
225 25
20

a=' ', b=0, c=0
A2::b=0
A1::a=0
0 0 0

a='d', b=10, c=9
A2::b=15
A1::a=5
361 225 25

5. #include <iostream.h>
   #include <string.h>
   class B1 {
       char* a;
   protected:
       B1(): a(0) {}
   public:
       ~B1() {delete []a; cout <<"B1 ";}
       void Output() {cout <<a <<endl;}
       void Seta(char* aa) {
           delete []a;
           a=new char[strlen(aa)+1];
           strcpy(a,aa);
       }
   };
   class B2: public B1 {
       char* b;
   public:
       B2(): b(0) {}
       ~B2() {delete []b; cout <<"B2 ";}
       void Output() {
           B1::Output();
           cout <<b <<endl;
       }
       void Setb(char* aa, char* bb) {
```

```

        Seta(aa);
        delete []b;
        b = new char[strlen(bb) + 1];
        strcpy(b,bb);
    }
};

void Input(B2 * &r, int n)
{
    r = new B2[n];
    char a[20], b[20];
    for(int i = 0; i < n; i++) {
        cout << "Input two strings: ";
        cin >> a >> b;
        r[i].Setb(a,b);
    }
}

void main()
{
    B2 * a = NULL;
    int n = 3;
    Input(a,n);
    for(int i = 0; i < n; i++)
        a[i].Output();
    delete []a;
    cout << endl << sizeof(B2) << endl;
}

```

在这个例子中包含有 B1 和 B2 两个类,每个类只含有一个数据成员,并且均为私有的字符指针。在 B1 中把无参函数定义为保护成员,它只能由本类及派生类调用,其他函数不能调用。在 B1 类中定义有三个公用成员函数,一个为析构函数,用以删除 a 所指向的动态字符空间;第二个为输出函数,用以输出 a 所指向的字符串;第三个为给 a 赋值的函数,它首先释放 a 所指向的存储空间,接着根据形参字符串的长度为 a 动态分配所指向的存储空间,最后把形参字符串拷贝到 a 所指向的存储空间中。在 B2 类中,定义有四个公用成员函数,一个为无参构造函数,它给字符指针 b 赋初值 0,即置空;第二个为析构函数,它删除 b 所指向的动态存储空间;第三个为输出函数,用以分别输出 a 和 b 指针所指向的字符串,由于 a 是基类 B1 的私有成员,在 B2 中不能直接访问,只能通过调用 B1 提供的输出函数 Output 输出 a 字符串,又由于 B2 中也定义有 Output 函数,所以在调用 B1 中的这个函数时必须在函数名前加上 B1 和类区分符;第四个为给 a 和 b 赋值的函数,它首先调用 Seta 函数(因其名字在 B2 类中惟一,所以加不加 B1 和类区分符限定均可)把 aa 字符串赋给 a 指针所指向的动态字符空间,接着把 bb 字符串赋给 b 指针所指向的动态字符空间。

程序中包含有一个 Input 函数,该函数中的第一条语句分配具有 n 个元素的类型为 B2 的动态数组,并将其首地址赋给引用形参 r,该动态数组的每一个元素将由自动调用的 B2 类的无参构造函数初始化,当然调用该构造函数时又将首先调用基类 B1 的无参构造函数,初始化属于基类的成员。第二条语句定义两个字符数组 a 和 b,用来保存下面循环中从键

盘上输入的字符串。第三条语句使循环体执行  $n$  次,每次从键盘的输入中顺序提取两个字符串并分别赋给  $a$  和  $b$ ,然后把它们作为实参用数组  $r$  中的相应元素去调用  $\text{Setb}$  成员函数,给该元素中的属于基类的私有成员  $a$  和在派生类中定义的私有成员  $b$  赋值。由于该函数使用的参数  $r$  为引用,所以在函数体中对  $r$  的赋值实际上是对调用该函数的对应实参的赋值,即利用实参可以得到在这个函数中动态分配的数组空间的首地址和数组中每个元素的值。

在主函数中定义有一个  $B2$  类的指针对象  $a$  并赋初值为空,用这个指针  $a$  和整型对象  $n$  作为实参去调用  $\text{Input}$  函数,该函数完成对动态分配的具有  $n$  个元素的  $a$  数组赋值的任务,接着主函数通过执行  $\text{for}$  循环输出  $a$  数组中的每个元素的值,然后使用  $\text{delete}$  语句删除(释放)指针  $a$  所指向的动态数组空间,由于动态数组中的每个元素为  $B2$  类对象,所以在释放前对于每一个元素都将自动调用  $B2$  类的析构函数,删除  $b$  指针所指向的动态字符串空间,又由于  $B2$  类包含有基类  $B1$ ,所以在执行  $B2$  类的析构函数的函数体之后,待返回之前,将自动调用基类  $B1$  的析构函数,删除  $a$  指针所指向的动态字符串空间,然后由基类  $B1$  的析构函数返回到  $B2$  的析构函数,再由  $B2$  的析构函数返回到主函数中正在执行的  $\text{delete}$  语句中,接着对下一个元素进行析构处理。

我们已经知道,构造函数的执行顺序是:先执行基类构造函数,接着执行类成员所属类的构造函数,最后执行自己的构造函数。而析构函数的执行顺序正好相反,它先执行自己的函数体,接着执行类成员所属类的析构函数,最后执行基类的析构函数。

程序输入和运行结果如下:

```
Input two strings: 1234 56789
Input two strings: qwe rtyui
Input two strings: ASDFG HJKL
1234
56789
qwe
rtyui
ASDFG
HJKL
B2 B1 B2 B1 B2 B1 8
```

### 9.1.3 应用举例

例 1. 编写一个程序计算出球、圆柱和圆锥的表面积和体积。

分析:由于计算它们都需要用到圆的半径,有时还可能用到圆的面积,所以可把圆定义为一个类。它包含的数据成员为半径,由于不需要作图,所以不需要定义圆心坐标。圆的半径应定义为保护属性,以便派生类能够继承和使用。圆类的公用函数为给半径赋初值的构造函数,计算圆的面积函数,也可以包含计算体积的函数,让其返回 0 即可,表示圆的体积为 0。定义好圆类后,再把球类、圆柱类和圆锥类定义为圆的派生类。在这些类中同样包含有新定义的构造函数、求表面积的函数和求体积的函数。另外在圆柱和圆锥类中应分别新定义一个表示其高度的数据成员。此题的完整程序如下:

```
#include <iostream.h>
#include <math.h>
```

```

const double PI = 3.1415926;
class Circle { // 圆类
protected:
    double r; // 半径
public:
    Circle(double radius = 0): r(radius) {}
    double Area() { // 计算圆的面积
        return PI * r * r;
    }
    double Volume() { // 计算圆的体积
        return 0;
    }
};

class Sphere: public Circle { // 球体类
public:
    Sphere(double radius = 0): Circle(radius) {}
    double Area() { // 计算球的表面积
        return 4 * PI * r * r;
        // 返回表达式可以用 4 * Circle::Area() 来代替
    }
    double Volume() { // 计算球的体积
        return 4 * PI * pow(r, 3) / 3;
        // pow(r, 3) 求出 r 的立方值, 此函数原型在 math.h 头文件中
    }
};

class Cylinder: public Circle { // 圆柱体类
    double h; // 高度
public:
    Cylinder(double radius = 0, double height = 0): Circle(radius) {
        h = height;
    }
    double Area() { // 计算圆柱体的表面积
        return 2 * PI * r * (r + h);
    }
    double Volume() { // 计算圆柱体的体积
        return PI * r * r * h;
        // 返回表达式可以用 Circle::Area() * h 来代替
    }
};

class Cone: public Circle { // 圆锥体类
    double h; // 高度
public:
    Cone(double radius = 0, double height = 0): Circle(radius) {
        h = height;
    }
    double Area() { // 计算圆锥体的表面积
        double l = sqrt(h * h + r * r); // sqrt 函数求出参数值的平方根。
        return PI * r * (r + l);
    }
};

```

```

    }
    double Volume() { // 计算圆锥体的体积
        return PI * r * r * h / 3;
    }
};

void main()
{
    Circle r1(2);
    Sphere r2(2);
    Cylinder r3(2,3);
    Cone r4(2,3);
    cout << "Circle: " << r1.Area() << ' ' << r1.Volume() << endl;
    cout << "Sphere: " << r2.Area() << ' ' << r2.Volume() << endl;
    cout << "Cylinder: " << r3.Area() << ' ' << r3.Volume() << endl;
    cout << "Cone: " << r4.Area() << ' ' << r4.Volume() << endl;
}

```

此程序运行结果如下:

```

Circle: 12.5664 0
Sphere: 50.2655 33.5103
Cylinder: 62.8319 37.6991
Cone: 35.2207 12.5664

```

例2. 假定居民的基本数据包括身份证号、姓名、性别和出生日期,而居民中的成年人又多两项数据:最高学历和职业,成人中的党员又多一项数据:党派类别。现要求建立3个类,让成人继承居民类,而党员类又继承成人,并要求在每个类中都提供有数据输入和输出的功能。

按题目要求定义三个类如下,仅供读者参考。

```

class People { // 居民类
    char num[19]; // 身份证号
    char name[11]; // 姓名
    int sex; // 性别
    char birth[11]; // 出生日期
public:
    void Input() {
        cout << "Input People data: " << endl;
        cin >> num >> name >> sex >> birth;
    }
    void Output() {
        cout << num << ' ' << name << ' ';
        cout << sex << ' ' << birth << endl;
    }
};

```

```

class Adult: public People { // 成人
    char sch[11]; // 最高学历
    char prof[11]; // 从事职业
public:

```



```

        void Input() {
            People::Input();
            cout << "Input sch & prof data:" << endl;
            cin >> sch >> prof;
        }
        void Output() {
            People::Output();
            cout << sch << " " << prof << endl;
        }
    };

class Party: public Adult { // 党员类
    char parties[15]; // 党派类别
public:
    void Input() {
        Adult::Input();
        cout << "Input parties data:" << endl;
        cin >> parties;
    }
    void Output() {
        Adult::Output();
        cout << parties << endl;
    }
};

```

## 9.2 类的虚函数与多态性

对于用户定义的每一个类型, C++ 只允许同一类型对象之间的赋值, 不允许不同类型对象之间的赋值, 若非要赋值不可, 则必须经过强制类型转换。当然, 不同用户类型之间的赋值通常也是没有意义的。但对于基类及其派生类来说, 情况有所不同, 由于派生类对象的首地址与所含的基类部分的首地址相同, 并且包含有基类的所有成员, 因此可以把派生类看做为基类的兼容类。所以 C++ 允许把派生类对象的地址赋给基类指针对象, 通过这个指针可以访问所指的基类对象, 即整个派生类对象中的基类部分; 也允许用派生类对象初始化基类的引用对象, 通过这个引用也可以访问所对应的基类对象; 还允许派生类对象向基类对象直接赋值, 它将把派生类对象中属于基类部分的值赋给基类对象中。例如在 9.1.3 小节的例 2 中, 假定 a1, a2 和 a3 分别是 People, Adult 和 Party 类的对象, p1, p2 和 p3 分别是这三个类的指针对象, s1 为基类 People 的引用对象, 则 a2 和 a3 均可以向 a1 赋值, 即把它们包含的 People 对象赋给 a1 中, a3 也可以向 a2 赋值, 即把 a3 中包含的 Adult 对象赋给 a2 中, a1, a2 和 a3 的地址, 以及 p2 和 p3 的值均可以赋给 p1, 通过 p1 可以访问所指的 People 类对象或派生类对象中的 People 对象, 同样可以用 a1, a2 和 a3 初始化引用 s1, 使 s1 为一个 People 类对象。

在一个含有基类和派生类的类系列中, 往往各个类中的相应的成员函数相同, 即具有相同的函数名、返回值类型和参数表, 但函数体可以不同, 即语义可以不同, 一般用每个函数实现与相应类有关的相似功能。如在 9.1.3 小节的例 1 中, 共有四个类, 其中圆为基类, 球、圆

柱和圆锥均为圆的派生类,它们的求表面积和体积的函数都相同,其函数名均为 Area 和 Volume,返回值类型为 double,参数表为空,它们分别用于实现与各自类有关的同一功能。在 9.1.3 小节的例 2 中,共有三个类,其中 People 为基类,Adult 为直接派生类,Party 为 People 的间接派生类,Adult 的直接派生类,这三个类的输入函数和输出函数都对应相同,用以实现与各自类有关的同一功能。

当一个派生类对象的地址赋给一个基类指针后,基类指针只能访问所属类的成员函数,不能访问到该派生类对象中与基类成员函数相同的成员函数。如对于 9.1.3 小节的例 1,将球类对象的地址赋给圆类指针对象后,利用该指针对象只能访问圆类中求表面积和体积的成员函数,不能访问到球类中求表面积和体积的成员函数。但在实际应用中,经常要求当把一个基类或派生类对象的地址赋给一个基类指针后,利用这个指针能够访问到该基类或派生类中的与基类成员函数相同的成员函数。如对于 9.1.3 节的例 1,要求当把一个圆类、球类、圆柱类或圆锥类对象的地址赋给一个圆类指针对象后,能够访问到相应类对象中的求表面积和体积的成员函数,求出相应类对象的表面积和体积。若能实现这一要求将给使用类编程带来极大的灵活和方便。C++ 语言提供了实现这一要求的手段,就是把基类和派生类中的相同函数都同时定义为虚函数。所谓虚函数就是在一个成员函数的函数类型标识符前加上代表虚函数的关键字 virtual 即可。C++ 语言还规定:当把基类中的一个函数定义为虚函数后,其直接派生类和间接派生类中的相同函数不管是否带有 virtual 关键字,则均被系统认为是虚函数。如对于 9.1.3 小节的例 1,若把圆类的求表面积的函数定义为虚函数后,则其他三个派生类中的求表面积函数也都自动成为虚函数。

当基类和派生类中的相同函数(但语义通常是不同的)定义为一组虚函数后,通过基类指针可以调用任一类中的虚函数执行(当然虚函数的执行和不规定为虚函数的执行情况完全是一样的),具体调用哪一个类中定义的虚函数则取决于程序运行时赋给基类指针的对象类型而定。如要求圆、球、圆柱和圆锥类对象中任一表面积时,只要将它们求各自表面积的函数定义为虚函数后,把相应类对象的地址赋给圆类指针对象(假定为 p),则通过 p->Area()的调用形式,就能够自动找到相应类中的虚函数执行,求出相应类对象的面积。这种通过调用基类的虚函数实际上能够调用一组虚函数中任一虚函数执行的技术称为多态性。

多态性除了通过虚函数和指针实现外,还可以通过虚函数和引用实现。当一个派生类对象初始化一个基类引用对象后,使用该引用调用在基类中规定为虚函数的成员函数时,也将能够通过动态定位(即在运行期间时的定位)调用初始化对象中的虚函数执行。初始化一个基类引用对象可以在定义时进行,也可以通过参数传递实现,当把一个实参传递给相应的引用对象时,就是用实参初始化该引用。

根据类的继承而定义的一组类中,除了构造函数不能定义为虚函数外,其他任一组相同的成员函数均可以定义为虚函数。特殊的一组类的析构函数也可以定义为虚函数,虽然它们的函数名是各不相同的,都是各自类的类名,但系统也认为它们是相同的成员函数,也能够被用户定义为虚函数使用。

另外,要实现类的多态性,还必须把所有派生类对基类的继承定义为公用继承,把每个类中的虚函数定义为具有公用访问权限的成员函数。

下面通过例子来说明虚函数和多态性的含义。

一个包含有虚函数的一组类和使用这组类中虚函数的主函数如下:

```
#include <iostream.h>
class X1 {
    int x;
public:
    X1(int xx=0) {x=xx;}
    virtual void Output() {
        cout <<"x=" <<x << endl;
    }
};
class Y1: public X1 {
    int y;
public:
    Y1(int xx=0, int yy=0): X1(xx) {y=yy;}
    virtual void Output() {
        X1::Output();
        cout <<"y=" <<y << endl;
    }
};
class Z1: public X1 {
    int z;
public:
    Z1(int xx=0, int zz=0): X1(xx) {z=zz;}
    virtual void Output() {
        X1::Output();
        cout <<"z=" <<z << endl;
    }
};

void main()
{
    X1 a(5); Y1 b(6,7); Z1 c(8,9);
    X1 * p[3] = {&a, &b, &c};
    for(int i=0; i<3; i++) {
        p[i]->Output(); cout << endl;
    }
}
```

在基类 X1 中把 Output() 函数定义为虚函数, 基类 X1 的公用继承类 Y1 和 Z1 中也都定义有 Output() 函数, 所以根据规定, 不管它们是否被定义为虚函数, 也都将自动成为虚函数。在主函数中定义有三个类对象 a, b 和 c, 它们分别属于 X1, Y1 和 Z1 类, 并且它们都带有初始化数据, 通过调用相应的构造函数可使其初始化。接着定义有一个基类 X1 指针数组, 它包含有三个元素, 并且分别被初始化为 a, b 和 c 的地址, 这样 p[0], p[1] 和 p[2] 就分别指向了基类 X1 的对象 a, 派生类 Y1 的对象 b 和派生类 Z1 的对象 c, 但注意实际上是指向各自对象中属于基类的那个部分。主函数最后是一个循环, 每次调用 p[i] 指针所属基类中的 Output() 成员函数, 但由于它在基类中被定义为一个虚函数, 所以实际被调用执行的是 p[i] 指针所指对象 (即最近被赋予地址值的那个对象或最近被赋予指针值的那个指针所指的对象)

的类中定义的虚函数,它并不一定是基类中的虚函数。

该程序的运行结果如下:

```
x = 5
```

```
x = 6
```

```
y = 7
```

```
x = 8
```

```
z = 9
```

若不把一组 `output()` 函数定义为虚函数(只要不把基类 `X1` 中的 `output()` 函数定义为虚函数即可),则程序运行中每次执行 `p[i] -> Output()` 语句时就不能进行动态定位,每次调用的必定是基类中的该成员函数的代码。取消虚函数定义后的程序运行结果如下:

```
x = 5
```

```
x = 6
```

```
x = 8
```

有时候需要定义基类纯粹是为定义派生类服务的,而不会被用来定义对象。如对于 9.1.3 小节的例 1,需要计算的是球、圆柱和圆锥体的表面积和体积,为此定义的基类为圆。为了使它们计算表面积和体积的函数均为虚函数,必须在圆类中也定义有相应的虚函数,但计算圆的表面积和体积没有实际意义,所以不需要用圆类来定义对象。

在实际应用中,当不需要用基类来定义对象(但可以用它定义指针和引用)时,可以把该基类定义为抽象类。定义一个类为抽象类的方法是把它的一些或全部虚函数定义为纯虚函数,纯虚函数必须是一个虚函数,它是一个只需进行虚函数声明,不需定义函数体的虚函数,并且在声明语句最后的分号前面加上赋值号与数值 0 做标记。例如,若将圆类定义中的求表面积和体积的函数均定义为纯虚函数,则如下所示:

```
class Circle { // 抽象基类
protected:
    double r; // 半径
public:
    Circle(double radius = 0): r(radius) {}
    virtual double Area() = 0; // 定义求物体面积的纯虚函数
    virtual double Volume() = 0; // 定义求物体体积的纯虚函数
};
```

此时的圆类被称为抽象类,它只能作为其他类的基类使用。

当把 9.1.3 小节例 1 程序中的圆类修改为上述抽象类,并把主函数做如下修改:

```
void Output(Circle* b[], int n) // 以类类型作为指针数组参数类型示例
{
    for(int i=0; i<n; i++)
        cout << b[i] -> Area() << ' ' << b[i] -> Volume() << endl;
}
```

```

void main()
{
    Sphere r1(2);
    Cylinder r2(2,3);
    Cone r3(2,3);
    Circle * a[3] = {&r1,&r2,&r3};
    Output(a,3);
}

```

则得到的程序运行结果如下:

```

50.2655 33.5103
62.8319 37.6991
35.2207 12.5664

```

### 9.3 类的静态成员

在一个类的定义中,若在一个成员定义的前面加上 `static` 关键字,则就声明了该成员为静态成员。静态成员也分为静态数据成员和静态成员函数两个方面,由于静态成员函数使用较少,这里只讨论静态数据成员的定义与使用。静态数据成员在生成的每个类对象中并不占有存储空间,而只是在每个类中分配有存储空间,并且该类的所有对象和外部函数都可以直接或间接地访问这个存储空间。当然静态数据成员也有访问属性,同非静态成员一样,本类成员函数能够直接访问具有任何访问属性的静态成员,但外部函数只能直接访问具有公用属性的静态成员,派生类中的成员函数能够直接访问所含基类中具有公共或保护属性的静态成员。

由于静态数据成员所占有的存储空间只与一个类有关,而与具体对象无关,所以既可以像使用成员操作符访问非静态成员那样访问静态成员,又可以在静态成员前加上类名和类区分符来访问一个类中的静态成员。如 `X` 表示一个类, `a` 为该类中一个具有公用访问属性的静态数据成员, `x` 为 `X` 类的一个对象, `y` 为 `X` 类的一个指针对象,则 `x.a`, `X::a` 和 `y -> a` 表示都指 `X` 类中的静态数据成员 `a`。

当在一个类中声明一个静态数据成员后,还需要在类外定义它并进行初始化,这样系统才能够为该静态成员分配对应的存储空间,并把初值赋给这个存储空间中,若没有对其进行初始化,则自动被赋初值 0。

请读者通过下面例子体会类中静态数据成员的声明、定义与使用。

```

1. #include <iostream.h>
   class XX {
       int a;
   public:
       static int b; // 声明 b 为公用属性的静态数据成员
       XX(int aa=0) {
           a=aa; b++;
       }
       int geta() {return a;}
   };

```

```
int XX::b=0; // 对 XX 类中的静态数据成员 b 进行定义和初始化
```

```
void main()
{
    cout << "XX::b=" << XX::b << endl;
    XX x(10), y(20);
    cout << "x.a, x.b=" << x.geta() << ", " << x.b << endl;
        // x.b 也可用 XX::b 代替
    cout << "y.a, y.b=" << y.geta() << ", " << y.b << endl;
        // y.b 也可用 XX::b 代替
}
```

该程序的运行结果如下:

```
XX::b=0
x.a, x.b=10, 2
y.a, y.b=20, 2
```

```
2. #include <iostream.h>
class YY {
    int a;
    static int b; // 声明 b 为具有私有属性的静态数据成员
public:
    YY(int aa=0) {a=aa;}
    int geta() {return a;}
    int getb() {return b;}
    void seta(int aa) {a=aa;}
    void setb(int bb) {b=bb;}
};
```

```
int YY::b=0; // 对 YY 私有静态数据成员 b 进行定义和初始化
```

```
void main()
{
    YY* pa = new YY(10);
    YY* pb = new YY(20);
    cout << "pa -> a, pa -> b=" << pa -> geta() << ", " << pa -> getb() << endl;
    cout << "pb -> a, pb -> b=" << pb -> geta() << ", " << pb -> getb() << endl;
    pa -> setb(5);
    pb -> seta(30); pb -> setb(15);
    cout << "pa -> a, pa -> b=" << pa -> geta() << ", " << pa -> getb() << endl;
    cout << "pb -> a, pb -> b=" << pb -> geta() << ", " << pb -> getb() << endl;
}
```

该程序运行结果如下:

```
pa -> a, pa -> b=10, 0
pb -> a, pb -> b=20, 0
pa -> a, pa -> b=10, 15
pb -> a, pb -> b=30, 15
```

类中的静态数据成员通常用来记录利用该类创建对象的个数等信息。

## 9.4 类模板

同函数模板是带类型参数的函数一样,类模板(又称模板类)也是带类型参数的类,同样结构模板是带类型参数的结构。由于类模板和结构模板的定义和使用方法相同,所以仅以类模板讨论之。

在一个类定义的前面加上 `template` 选项,在类定义中使用该选项中提供的类型参数,则就实现了一个类模板的定义。`template` 选项以关键字 `template` 开始,后跟一对尖括号,尖括号内为类型参数表,类型参数表中给出一个或多个类型参数,当多于一个时,各类型参数之间要用逗号分开,每个类型参数为一个用户定义的标识符,用它作为一种形式类型,可以在类定义体内作为一种已定义的类型使用,另外每个类型参数还必须以关键字 `class` 为前导。

下面通过例子说明类模板的定义和使用。

请看下面定义的一个类模板。

```
template < class Type > // Type 为类型参数
class TwoNum { // 由两个 Type 类型的数据成员构成的类模板
    Type a;
    Type b;
public:
    TwoNum() {}; // 无参构造函数
    TwoNum(Type aa, Type bb): a(aa), b(bb) {} // 带参构造函数
    int Compare() { // 比较 a 和 b 的大小
        if(a > b) return 1;
        else if(a == b) return 0;
        else return -1;
    }
    Type Maximum() { // 返回 a 和 b 中的最大值
        return (a > b)? a:b;
    }
    Type Minimum() { // 返回 a 和 b 中的最小值
        return (a > b)? b:a;
    }
    void Setab(Type& aa, Type& bb) { // 给 a 和 b 赋值
        a = aa; b = bb;
    }
    Type geta() {return a;} // 返回 a 的值
    Type getb() {return b;} // 返回 b 的值
};
```

这个类模板定义有一个类型参数 `Type`,它的作用域为所在的类,在类定义体中使用它就如同使用其他已定义的类型一样。该类型中定义的两个数据成员都是 `Type` 类型的,定义的带参构造函数的参数和 `Setab` 函数的参数也都是 `Type` 类型的,另外还有四个函数的返回值是 `Type` 类型的。

当类模板中的成员函数在类外定义时,则它必须定义为一个函数模板的形式,即在函数

定义的前面要带有 `template` 选项,并且在类区分符前面使用的类名要后跟一对尖括号,尖括号内给出类型参数,但此时的每个类型参数前不能带有 `class` 关键字。

例如,若把 `TwoNum` 类中的带参构造函数写成类外定义时,则为:

```
template < class Type >
TwoNum < Type > :: TwoNum( Type aa, Type bb): a(aa), b(bb) {}
```

若把 `TwoNum` 类中的 `Compare` 函数写成类外定义时,则为:

```
template < class Type >
int TwoNum < Type > :: Compare() {
    if(a > b) return 1;
    else if(a == b) return 0;
    else return -1;
}
```

使用类模板定义对象时,类名标识符后要带有一对尖括号,尖括号内为类型实参表,实参个数要与类模板定义中类型参数表给出的参数个数相同,当然类型实参表所给出的实参必须是已定义的类型。当程序编译中遇到使用有类模板时,系统将根据类型实参表中所给的类型去替换类模板定义中的相应形式类型参数,从而生成一个真实的类,称它为类模板的一个实例。使用类模板的类名后跟用一对尖括号括起来的类型实参表就是将类模板实例化的过程,类模板只有被实例化后才能定义对象。如:

```
TwoNum < int > a(5,6);
```

首先根据 `TwoNum` 类模板生成一个 `Type` 被替换为 `int` 的类,然后定义这个类的一个对象 `a`,并利用所给的实参对其进行初始化,使数据成员 `a` 和 `b` 的值分别为 5 和 6。

在下面的主函数中使用了上面定义的 `TwoNum` 类。

```
#include <iostream.h>
#include "Frunction.h" // 包含有分数类定义的头文件
template < class Type > class TwoNum {}; // 类定义体在此省略
void main()
{
    TwoNum < int > x(4,8);
    cout << x.Compare() << ' ' << x.Maximum() << ' ' << x.Minimum() << endl;
    char ch='x';
    TwoNum < char > y(ch,'x');
    cout << y.Compare() << ' ' << y.Maximum() << ' ' << y.Minimum() << endl;
    Frunction f1(5,4),f2(4,5);
    TwoNum < Frunction > a(f1,f2);
    cout << a.Compare() << ' ' << a.Maximum() << ' ' << a.Minimum() << endl;
}
```

主函数中的第一条语句对类模板进行整型实例化并定义对象 `x`,第四条语句对类模板进行字符类型实例化并定义对象 `y`,第六条语句定义两个分数类类型的对象 `f1` 和 `f2`,第七条语句对类模板进行分数类类型实例化并定义对象 `a`。该程序运行结果如下:

```
-184
0 x x
• 284 •
```



## 习题九

### (一) 填空题

1. 对基类数据成员的初始化是通过执行派生类构造函数中的\_\_\_\_\_来实现的。
2. 在一个派生类中,对基类成员、类对象成员和非类对象成员的初始化次序是先\_\_\_\_\_,后\_\_\_\_\_,最后为\_\_\_\_\_。
3. 当撤消一个含有基类和类对象成员的派生类对象时,将首先完成\_\_\_\_\_的析构函数定义体的执行,接着完成\_\_\_\_\_的析构函数定义体的执行,最后完成\_\_\_\_\_的析构函数定义体的执行。
4. 假定一个类 AB 中有一个静态整型成员 bb,在类外为它进行定义并初始化为 0 时,所使用的语句为\_\_\_\_\_。
5. 假定类 AB 中有一个公用属性的静态数据成员 bb,在类外不通过对象名访问该成员 bb 的写法为\_\_\_\_\_。

### (二) 写出下列程序运行后的输出结果

1. 

```
#include<iostream.h>
class A {
    int a;
public:
    A(int aa=0): a(aa) {
        cout <<"Constructor A!" <<a << endl;
    }
};
class B: public A {
    int b;
public:
    B(int aa, int bb): A(aa), b(bb) {
        cout <<"Constructor B!" <<b << endl;
    }
};
void main() {
    B x(2,3), y(4,5);
}
```
2. 

```
#include<iostream.h>
class A {
    int a;
public:
    A(int aa=0) {a=aa;}
    ~A() {cout <<"Destructor A!" <<a << endl;}
};
```

```

class B: public A {
    int b;
public:
    B(int aa = 0, int bb = 0): A(aa) {b = bb;}
    ~B() {cout << "Destructor B!" << b << endl;}
};

void main() {
    B x(5), y(6,7);
}

```

### 3. #include <iostream.h>

```

class AX {
    int x;
public:
    AX(int xx = 0): x(xx) {
        cout << "AX constructor." << endl;
    }
    ~AX() {
        cout << "AX destructor." << endl;
    }
    void Output() {
        cout << x << ' ';
    }
    int Get() {return x;}
};

class BX: public AX {
    int y;
    AX z;
public:
    BX(int xx = 0, int yy = 0): AX(xx), y(yy), z(xx + yy) {
        cout << "BX constructor." << endl;
    }
    ~BX() {
        cout << "BX destructor." << endl;
    }
    void Output() {
        AX::Output(); cout << Get() << ' ';
        cout << y << ' ' << z.Get() << endl;
    }
};

void main()
{
    BX a(5), b(10,20);
    a.Output();
    b.Output();
}

```

### 4. #include <iostream.h>

```

class AY {

```

```

protected:
    int a,b;
public:
    AY(int aa=0, int bb=0) {
        a=aa; b=bb;
    }
    virtual void Compute() {
        cout << a << '+' << b << '=' << a+b << endl;
    }
};

class BY: public AY {
public:
    BY(int aa=0, int bb=0): AY(aa,bb) {}
    void Compute() {
        cout << a << '-' << b << '=' << a-b << endl;
    }
};

class CY: public BY {
public:
    CY(int aa=0, int bb=0): BY(aa,bb) {}
    void Compute() {
        cout << a << '*' << b << '=' << a*b << endl;
    }
};

class DY: public AY {
public:
    DY(int da=0, int db=0): AY(da,db) {}
    void Compute() {
        if(b!=0)
            cout << a << '/' << b << '=' << a/b << endl;
        else
            cout << "divisor is zero!" << endl;
    }
};

void main()
{
    int n=10, m=5;
    AY ay(n,m); BY by(n,m);
    CY cy(n,m); DY dy(n,m);
    AY * a[4] = {&ay, &by, &cy, &dy};
    for(int i=0; i<4; i++)
        a[i] -> Compute();
    AY &ax = cy; ax.Compute();
    AY aa = cy; aa.Compute();
}

```

5. #include<iostream.h>

```

#include< string.h>
#include< stdlib.h>

const int MaxSize=20;

struct AA {
    char a[10];
    int b;
    int operator > (AA& x) {return (b>x.b)? 1:0;}
    int operator < (AA& x) {return (b<x.b)? 1:0;}
    void operator += (AA& x) {b += x.b;}
    float operator/(int n) {return float(b)/n;}
};

ostream& operator << (ostream& ostr, AA& x) {
    ostr << x.a << " " << x.b;
    return ostr;
}

template< class DataType>
class List {
    DataType list[MaxSize];
    int n;
public:
    List() {n=0;}
    List(DataType a[], int);
    void OutMax();
    void OutMin();
    void OutMean();
};

template< class DataType>
List< DataType>::List(DataType a[], int nn) {
    if(nn <= 0 || n>MaxSize) {
        cerr << "the value of n not correct!" << endl;
        exit(1);
    }
    n=nn;
    for(int i=0; i<n; i++)
        list[i]=a[i];
}

template< class DataType>
void List< DataType>::OutMax() {
    int k=0;
    for(int i=1; i<n; i++)
        if(list[i]>list[k]) k=i;
    cout << "Maximum: " << list[k] << endl;
}

template< class DataType>

```

```

void List<DataType>::OutMin() {
    int k=0;
    for(int i=1; i<n; i++)
        if(list[i]<list[k]) k=i;
    cout <<"Minimum: " <<list[k]<<endl;
}

template< class DataType>
void List<DataType>::OutMean() {
    DataType s = list[0];
    for(int i=1; i<n; i++)
        s += list[i];
    cout <<"Mean: " << s/n << endl;
}

void main()
{
    int a1[6] = {4,7,6,2,5,9};
    AA a2[4] = {{"xok", 46}, {"wr", 44}, {"nch", 39}, {"shyf", 48}};
    List<int> b1(a1,6);
    b1.OutMax(); b1.OutMin(); b1.OutMean();
    List<AA> b2(a2,4);
    b2.OutMax(); b2.OutMin(); b2.OutMean();
}

```

## 第十章 C++ 流

### 10.1 C++ 流的概念

在C++语言中,数据的输入和输出(简称为I/O)包括对标准输入设备键盘和标准输出设备显示器、对在外存磁盘上的文件和对内存中指定的字符串存储空间(当然可用该空间存储任何信息)进行输入输出这三个方面。对标准输入设备和标准输出设备的输入输出简称为标准I/O,对在外存磁盘上文件的输入输出简称为文件I/O,对内存中指定的字符串存储空间的输入输出简称为串I/O。

C++语言系统为实现数据的输入和输出定义了一个庞大的类库,它包括的类主要有ios, istream, ostream, iostream, ifstream, ofstream, fstream, istrstream, ostrstream, strstream等,其中ios为根基类,其余都是它的直接或间接派生类。类库中包含的所有类以及继承关系如图10-1所示。

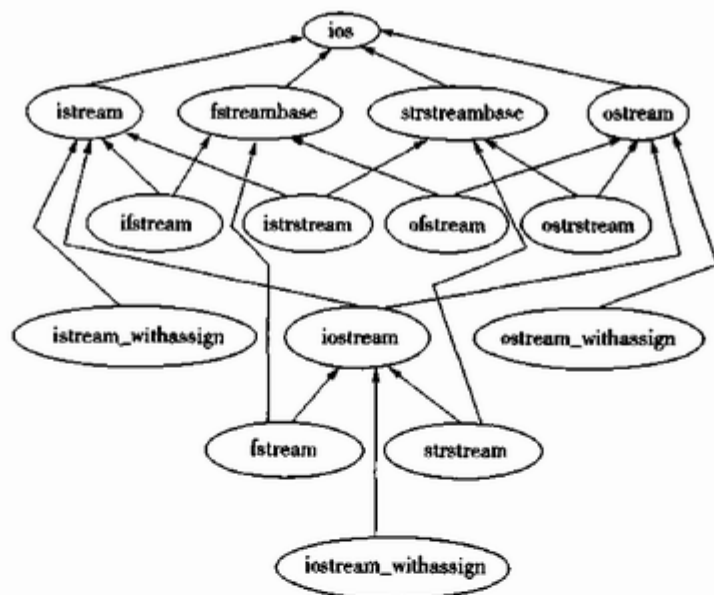


图 10-1 C++ 所有 I/O 类的继承关系图

从图中可以清楚地看出,根基类ios直接派生四个类:输入流类istream、输出流类ostream、文件流基类fstreambase和字符串流基类strstreambase。输入文件流类ifstream同时继承了输入流类和文件流基类(当然对于根基类是间接继承);输出文件流类ofstream同时继承了输出流类和文件流基类;输入字符串流类istrstream同时继承了输入流类和字符串流基

类,输出字符串流类 `ostream` 同时继承了输出流类和字符串流基类;输入输出流类 `iostream` 同时继承了输入流类和输出流类;输入输出文件流类 `fstream` 同时继承了输入输出流类和文件流基类;输入输出字符串流类 `stringstream` 同时继承了输入输出流类和字符串流基类。

“流”就是“流动”,是物质从一处向另一处流动的过程。C++ 流是指信息从外部输入设备(如键盘和磁盘)向计算机内部(即内存)输入和从内存向外部输出设备(如显示器和磁盘)输出的过程,这种输入输出过程被形象地比喻为“流”。为了实现信息的内外流动,C++ 系统定义了 I/O 类库,其中的每一个类都称做相应的流或流类,用以完成某一方面的功能。根据一个流类定义的对象也时常被称为流。如根据文件流类 `fstream` 定义的一个对象 `fio`,可称为 `fio` 流或 `fio` 文件流,用它可以同磁盘上一个文件相联系,实现对该文件的输入和输出,`fio` 就等同于与之相联系的文件。

C++ 系统中的 I/O 类库,其所有类被包含在 `iostream.h`, `fstream.h` 和 `sstream.h` 这三个系统头文件中,各头文件包含的类如下:

`iostream.h` 包含有: `ios`, `istream`, `ostream`, `istream_withassign`, `ostream_withassign`, `ostream_withassign` 等。

`fstream.h` 包含有: `fstream`, `ifstream`, `ofstream` 和 `fstreambase`,以及 `iostream.h` 中的所有类。

`sstream.h` 包含有: `stringstream`, `istringstream`, `ostringstream` 和 `stringstreambase`,以及 `iostream.h` 中的所有类。

在一个程序或一个编译单元(即一个程序文件)中当需要进行标准 I/O 操作时,则必须包含头文件 `iostream.h`,当需要进行文件 I/O 操作时,则必须包含头文件 `fstream.h`,同样,当需要进行串 I/O 操作时,则必须包含头文件 `sstream.h`。在一个程序或编译单元中包含一个头文件的命令格式为“`#include <头文件名>`”。当然若头文件是用户建立的,则头文件名的两侧不是使用尖括号,而是使用双引号。当系统编译一个 C++ 文件对 `#include` 命令进行处理时,是把该命令中指定的文件中的全部内容嵌入到该命令的位置,然后再编译整个 C++ 文件生成相应的目标代码文件。

C++ 不仅定义有现成的 I/O 类库供用户使用,而且还为用户进行标准 I/O 操作定义了四个类对象,它们分别是 `cin`, `cout`, `cerr` 和 `clog`。其中 `cin` 为 `istream_withassign` 流类的对象,代表标准输入设备键盘,也称为 `cin` 流或标准输入流。后三个为 `ostream_withassign` 流类的对象,`cout` 代表标准输出设备显示器,也称为 `cout` 流或标准输出流。`cerr` 和 `clog` 含义相同,均代表错误信息输出设备显示器。因此当进行键盘输入时使用 `cin` 流。当进行显示器输出时使用 `cout` 流。当进行错误信息输出时使用 `cerr` 或 `clog`。

在 `istream` 输入流类中定义有对右移操作符 `>>` 重载的一组公用成员函数,函数的具体声明格式为:

```
istream& operator >> (简单类型标识符 &);
```

简单类型标识符可以为 `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`, `char*`, `signed char*`, `unsigned char*` 之中的任何一种,对于每一种类型都对应着一个右移操作符重载函数。由于右移操作符重载用于给变量输入数据的操作,所以又称为提取操作符,即从流中提取出数据赋给变量。

当系统执行 `cin >> x` 操作时,将根据实参 `x` 的类型调用相应的提取操作符重载函数,把

x 引用传送给对应的形参。接着从键盘的输入中读入一个值并赋给 x (因形参是 x 的别名) 后, 返回 cin 流, 以便继续使用提取操作符为下一个变量输入数据。

当从键盘上输入数据时, 只有当输入完数据并按下回车键后, 系统才把该行数据存入到键盘缓冲区, 供 cin 流顺序读取给变量。还有, 从键盘上输入的每个数据之间必须用空格或回车符分开, 因为 cin 为一个变量读入数据时是以空格或回车符作为其结束标志的。

当 cin >> x 操作中的 x 为字符指针类型时, 则要求从键盘的输入中读取一个字符串, 并把它赋值给 x 所指向的存储空间中, 若 x 没有事先指向一个允许写入信息的存储空间, 则无法完成输入操作。另外从键盘上输入的字符串, 其两边不能带有双引号定界符, 若带有只作为双引号字符看待。对于输入的字符也是如此, 不能带有单引号定界符。

在 ostream 输出流类中定义有对左移操作符 << 重载的一组公用成员函数, 函数的具体声明格式为:

```
ostream& operator << (简单类型标识符);
```

简单类型标识符除了与在 istream 流类中声明右移操作符重载函数给出的所有简单类型标识符相同以外, 还增加一个 void \* 类型, 用于输出任何指针 (但不能是字符指针, 因为它将被作为字符串处理, 即输出所指向存储空间中保存的一个字符串) 的值。由于左移操作符重载用于向流中输出表达式的值, 所以又称为插入操作符。如当输出流是 cout 时, 则就把表达式的值插入到显示器上, 即输出到显示器显示出来。

当系统执行 cout << x 操作时, 首先根据 x 值的类型调用相应的插入操作符重载函数, 把 x 的值按值传送给对应的形参。接着执行函数体, 把 x 的值 (亦即形参的值) 输出到显示器屏幕上, 从当前屏幕光标位置起显示出来。然后返回 cout 流, 以便继续使用插入操作符输出下一个表达式的值。当使用插入操作符向一个流输出一个值后, 再输出下一个值时将被紧接着放在上一个值的后面。所以为了让流中前后两个值分开, 可以在输出一个值之后接着输出一个空格, 或一个换行符, 或其他所需要的字符或字符串。

## 10.2 输入输出格式控制

### 10.2.1 ios 类中的枚举常量

在根基类 ios 中定义有三个用户需要使用的枚举类型, 由于它们是在公用成员部分定义的, 所以其中的每个枚举类型常量在加上 ios:: 前缀后都可以为本类成员函数和所有外部函数访问。在三个枚举类型中有一个无名枚举类型, 其中定义的每个枚举常量都是用于设置控制输入输出格式的标记使用的。该枚举类型定义如下:

```
enum {skipws, left, right, internal, dec, oct, hex, showbase,
      showpoint, uppercase, showpos, scientific, fixed, unitbuf, stdio
};
```

各枚举常量的含义如下:

**skipws**



利用它设置对应标志后,从流中输入数据时跳过当前位置及后面的所有连续的空白字符,从第一个非空白字符起读数,否则不跳过空白字符。空格、制表符'\t'、回车符'\r'和换行符'\n'统称为空白符。默认为设置。

#### **left, right, internal**

left 在指定的域宽内按左对齐输出, right 按右对齐输出, 而 internal 使数值的符号按左对齐、数值本身按右对齐输出。域宽内剩余的字符位置用填充符填充。默认为 right 设置。在任一时刻只有一种有效。

#### **dec, oct, hex**

设置 dec 对应标志后,使以后的数值按十进制输出,设置 oct 后按八进制输出,而设置 hex 后则按十六进制输出。默认为 dec 设置。

#### **showbase**

设置对应标志后使数值输出的前面加上“基指示符”,八进制数的基指示符为数字 0,十六进制数的基指示符为 0x,十进制数没有基指示符。默认为不设置,即在数值输出的前面不加基指示符。

#### **showpoint**

强制输出的浮点数中带有小数点和小数尾部的无效数字 0。默认为不设置。

#### **uppercase**

使输出的十六进制数和浮点数中使用的字母为大写。默认为不设置,即输出的十六进制数和浮点数中使用的字母为小写。

#### **showpos**

使输出的正数前带有正号“+”。默认为不设置,即输出的正数前不带任何符号。

#### **scientific, fixed**

进行 scientific 设置后使浮点数按科学表示法输出,进行 fixed 设置后使浮点数按定点表示法输出。只能任设其一。默认时由系统根据输出的数值选用合适的表示输出。

#### **unitbuf, stdio**

这两个常量很少使用,所以不予介绍。

在 ios 中定义的第二个枚举类型为:

```
enum open_mode {in, out, ate, app, trunc, nocreate, noreplace, binary};
```

其中的每个枚举常量规定一种文件打开的方式,在定义文件流对象和打开文件时使用。

在 ios 中定义的第三个枚举类型为:

```
enum seek_dir {beg, cur, end};
```

其中的每个枚举常量用于对文件指针的定位操作上。

### **10.2.2 ios 类中的成员函数**

ios 类提供成员函数对流的状态进行检测和进行输入输出格式控制等操作,每个成员函数的声明格式和简要说明如下:

```
int bad(); // 操作出错时返回非 0 值。
```

`int eof();` // 读取到流中最后的文件结束符时返回非 0 值。  
`int fail();` // 操作失败时返回非 0 值。  
`void clear();` // 清除 bad, eof 和 fail 所对应的标志状态,使之恢复为正常状态  
// 值 0,使 good 标志状态恢复为 1。  
`char fill();` // 返回当前使用的填充字符。  
`char fill(char c);` // 重新设置流中用于输出数据的填充字符为 c 的值,返回此  
// 前的填充字符。系统预设置填充字符为空格。  
`long flags();` // 返回当前用于 I/O 控制的格式状态字。  
`long flags(long f);` // 重新设置格式状态字为 f 的值,返回此前的格式状态字。  
`int good();` // 操作正常时返回非 0 值,当操作出错、失败和读到文件结束符时  
// 均为不正常,则返回 0。  
`int precision();` // 返回浮点数输出精度,即输出的有效数字的位数。  
`int precision(int n);` // 设置浮点数的输出精度为 n,返回此前的输出精度。  
// 系统预设置的输出精度为 6,即输出的浮点数最多  
// 具有 6 位有效数字。  
`int rdstate();` // 操作正常时返回 0,否则返回非 0 值,它与 good()正好相反。  
`long setf(long f);` // 根据参数 f 设置相应的格式化标志,返回此前的设置。  
// 该参数 f 所对应的实参为无名枚举类型中的枚举常量(又  
// 称格式化常量),可以同时使用一个或多个常量,每两个  
// 常量之间要用按位或操作符连接。如当需要左对齐输出,  
// 并使数值中的字母大写时,则调用该函数的实参为 `ios::left | ios::uppercase`。  
`long unsetf(long f);` // 根据参数 f 清除相应的格式化标志,返回此前的设置。  
// 如要清除此前的左对齐输出设置,恢复默认的右对齐输出  
// 设置,则调用该函数的实参为 `ios::left`。  
`int width();` // 返回当前的输出域宽。若返回数值 0 则表明没有为刚才输出的  
// 数值设置输出域宽,输出域宽是指输出的值在流中所占有的字节数。  
`int width(int w);` // 设置下一个数据值的输出域宽为 w,返回为输出上一个数  
// 据值所规定的域宽,若无规定则返回 0。注意:此设置不  
// 是一直有效,而只是对下一个输出数据有效。

因为所有 I/O 流类都是 `ios` 的派生类,所以它们的对象都可以调用 `ios` 类中的成员函数和使用 `ios` 类中的格式化常量进行输入输出格式控制。下面以标准输出流对象 `cout` 为例说明输出的格式化控制。

程序 10-1:

```

#include <iostream.h>
void main()
{
    int x = 30, y = 300, z = 1024;
    cout << x << ' ' << y << ' ' << z << endl; // 按十进制输出
    cout.setf(ios::oct); // 设置为八进制输出
}

```

```

cout << x << ' ' << y << ' ' << z << endl; // 按八进制输出
cout.unsetf(ios::oct);
    // 取消八进制输出设置,恢复按十进制输出
cout.setf(ios::hex); // 设置为十六进制输出
cout << x << ' ' << y << ' ' << z << endl; // 按十六进制输出
cout.setf(ios::showbase | ios::uppercase);
    // 设置基指示符输出和数值中的字母大写输出
cout << x << ' ' << y << ' ' << z << endl;
cout.unsetf(ios::showbase | ios::uppercase);
    // 取消基指示符输出和数值中的字母大写输出
cout << x << ' ' << y << ' ' << z << endl;
cout.unsetf(ios::hex);
    // 取消十六进制输出设置,恢复按十进制输出
cout << x << ' ' << y << ' ' << z << endl;
}

```

此程序的运行结果如下:

```

30 300 1024
36 454 2000
1e 12c 400
0X1E 0X12C 0X400
1e 12c 400
30 300 1024

```

#### 程序 10-2:

```

#include <iostream.h>
void main()
{
    int x = 468;
    double y = -3.425648;
    cout << "x = ";
    cout.width(10); // 设置输出下一个数据的域宽为 10
    cout << x; // 按默认的右对齐输出, 剩余位置填充空格字符
    cout << "y = ";
    cout.width(10); // 设置输出下一个数据的域宽为 10
    cout << y << endl;
    cout.setf(ios::left); // 设置按左对齐输出
    cout << "x = ";
    cout.width(10);
    cout << x;
    cout << "y = ";
    cout.width(10);
    cout << y << endl;
    cout.fill('*'); // 设置填充字符为 '*'
    cout.precision(3); // 设置浮点数输出精度为 3
    cout.setf(ios::showpos); // 设置正数的正号输出
    cout << "x = ";
    cout.width(10);
    cout << x;
    cout << "y = ";

```

```

        cout.width(10);
        cout <<y <<endl;
    }

```

此程序运行结果如下：

```

x=      468y=   -3.42565
x=468      y=   -3.42565
x=+468*****y=-3.43*****

```

程序 10-3:

```

#include <iostream.h>
void main()
{
    float x=25, y=-4.762;
    cout <<x <<' '<<y <<endl;
    cout.setf(ios::showpoint); // 强制显示小数点和无效 0
    cout <<x <<' '<<y <<endl;
    cout.unsetf(ios::showpoint); // 恢复默认输出
    cout.setf(ios::scientific); // 设置按科学表示法输出
    cout <<x <<' '<<y <<endl;
    cout.setf(ios::fixed); // 设置按定点表示法输出
    cout <<x <<' '<<y <<endl;
}

```

程序运行结果如下：

```

25 -4.762
25.0000 -4.76200
2.500000e+001 -4.762000e+000
25 -4.762

```

### 10.2.3 格式控制操纵符

数据输入输出的格式控制还有更简便的形式,就是使用系统头文件 `iomanip.h` 中提供的操纵符。使用这些操纵符不需要调用成员函数,只要把它们作为插入操作符 `<<` (个别作为提取操作符 `>>`) 的输出对象即可。这些操纵符及功能如下:

```

dec    // 转换为按十进制输出整数,它也是系统预置的进制。
oct    // 转换为按八进制输出整数。
hex    // 转换为按十六进制输出整数。
ws     // 从输入流中读取空白字符。
endl   // 输出换行符'\n'并刷新流。刷新流是指把流缓冲区的内容立即写入到对
        // 应的物理设备上。
ends   // 输出一个空字符'\0'。
flush  // 只刷新一个输出流。
setiosflags(long f) // 设置 f 所对应的格式化标志,功能与 setf(long f)
                    // 成员函数相同,当然输出该操纵符后返回的是一个

```

```

// 输出流。如采用标准输出流 cout 输出它时,则返回
// cout。对于输出每个操纵符后也都是如此,即返回
// 输出它的流,以便向流中继续插入下一个数据。
resetiosflags(long f) // 清除 f 所对应的格式化标志,功能与 unsetf(long f)
// 成员函数相同。当然输出后返回一个流。
setfill(int c) // 设置填充字符为 ASCII 码为 c 的字符。
setprecision(int n) // 设置浮点数的输出精度为 n。
setw(int w) // 设置下一个数据的输出域宽为 w。

```

在上面的操纵符中,dec, oct, hex, endl, ends, flush 和 ws 除了在 iomanip.h 中有定义外,在 iostream.h 中也有定义。所以当程序或编译单元中只需要使用这些不带参数的操纵符时,可以只包含 iostream.h 文件,而不需要包含 iomanip.h 文件。

下面以标准输出流对象 cout 为例,说明使用操作符进行的输出格式化控制。

程序 10-4:

```

#include <iostream.h>
// 因 iomanip.h 中包含有 iostream.h,所以该命令可省略
#include <iomanip.h>
void main()
{
    int x=30, y=300, z=1024;
    cout << x << ' ' << y << ' ' << z << endl; // 按十进制输出
    cout << oct << x << ' ' << y << ' ' << z << endl; // 按八进制输出
    cout << hex << x << ' ' << y << ' ' << z << endl; // 按十六进制输出
    cout << setiosflags(ios::showbase | ios::uppercase);
        // 设置基指示符和数值中的字母大写输出
    cout << x << ' ' << y << ' ' << z << endl; // 仍按十六进制输出
    cout << resetiosflags(ios::showbase | ios::uppercase);
        // 取消基指示符和数值中的字母大写输出
    cout << x << ' ' << y << ' ' << z << endl; // 仍按十六进制输出
    cout << dec << x << ' ' << y << ' ' << z << endl; // 按十进制输出
}

```

此程序的功能和运行结果都与程序 10-1 完全相同。

程序 10-5:

```

#include <iostream.h>
#include <iomanip.h>
void main()
{
    int x=468;
    double y=-3.425648;
    cout << "x=" << setw(10) << x;
    cout << "y=" << setw(10) << y << endl;
    cout << setiosflags(ios::left); // 设置按左对齐输出
    cout << "x=" << setw(10) << x;
    cout << "y=" << setw(10) << y << endl;
    cout << setfill('*'); // 设置填充字符为 '*'
    cout << setprecision(3); // 设置浮点数输出精度为 3
}

```

```

    cout << setiosflags(ios::showpos); // 设置正数的正号输出
    cout << "x=" << setw(10) << x;
    cout << "y=" << setw(10) << y << endl;
    cout << resetiosflags(ios::left | ios::showpos);
    cout << setfill(' ');
}

```

此程序的功能和运行结果完全与程序 10-2 相同。

程序 10-6:

```

#include <iomanip.h>
void main()
{
    float x=25, y=-4.762;
    cout << x << ' ' << y << endl;
    cout << setiosflags(ios::showpoint);
    cout << x << ' ' << y << endl;
    cout << resetiosflags(ios::showpoint);
    cout << setiosflags(ios::scientific);
    cout << x << ' ' << y << endl;
    cout << setiosflags(ios::fixed);
    cout << x << ' ' << y << endl;
}

```

此程序的功能和运行结果也完全与程序 10-3 相同。

## 10.3 文件操作

### 10.3.1 文件的概念

以前进行的输入输出操作都是在键盘和显示器上进行的,通过键盘向程序输入待处理的数据,通过显示器输出程序运行过程中需要告诉用户的信息。键盘是C++系统中的标准输入设备,用 cin 流表示。显示器是C++系统中的标准输出设备,用 cout 流表示。

数据的输入和输出除了可以在键盘和显示器上进行之外,还可以在磁盘上进行。磁盘是外部存储器,它能够永久保存信息,并能够被重新读写和携带使用。所以若用户需要把信息保存起来,以便下次使用,则必须把它存储到外存磁盘上。

在磁盘上保存的信息是按文件的形式组织的,每个文件都对应一个文件名,并且属于某个物理盘或逻辑盘的目录层次结构中一个确定的目录之下。一个文件名由文件主名和扩展名两部分组成,它们之间用圆点(即小数点)分开,扩展名可以省略,当省略时也要省略掉前面的圆点。文件主名是由用户命名的一个有效的C++标识符,为了同其他软件系统兼容,一般让文件主名为不超过8个有效字符的标识符,同时为了便于记忆和使用,最好使文件主名的含义与所存的文件内容相一致。文件扩展名也是由用户命名的、1~3个字符组成的、有效的C++标识符,通常用它来区分文件的类型。如在C++系统中,用扩展名 h 表示头文件,用扩展名 cpp 表示程序文件,用 obj 表示程序文件被编译后生成的目标文件,用 exe 表示

连接整个程序中所有目标文件后生成的可执行文件。对于用户建立的用于保存数据的文件,通常用 `.dat` 表示扩展名,若它是由字符构成的文本文件则也用 `.txt` 作为扩展名,若它是由字节构成的、能够进行随机存取的内部格式文件则可用 `.ran` 表示扩展名。

在 C++ 程序中使用的保存数据的文件按存储格式分为两种类型,一种为字符格式文件,简称字符文件,另一种为内部格式文件,简称字节文件。字符文件又称 ASCII 码文件或文本文件,字节文件又称二进制文件。在字符文件中,每个字节单元的内容为字符的 ASCII 码,被读出后能够直接送到显示器或打印机上显示或打印出对应的字符,供人们直接阅读。在字节文件中,文件内容是数据的内部表示,是从内存中直接复制过来的。当然对于字符信息,数据的内部表示就是 ASCII 码表示,所以在字符文件和在字节文件中保存的字符信息没有差别,但对于数值信息,数据的内部表示和 ASCII 码表示截然不同,所以在字符文件和在字节文件中保存的数值信息也截然不同。如对于一个短整型数 1069,它的内部表示占有两个字节,对应的十六进制编码为 04 2D,其中 04 为高字节值,2D 为低字节值;若用 ASCII 码表示则为 4 个字节,每个字节依次为 1069 中每个字符的 ASCII 码,对应的十六进制编码为 31 30 36 39。当从内存向字符文件输出数值数据时需要自动转换成它的 ASCII 码表示,相反,当从字符文件向内存输入数值数据时也需要自动将它转换为内部表示,而对于字节文件的输入输出则不需要转换,仅是内外存信息的直接拷贝,显然比字符文件的输入输出要快得多。所以当建立的文件主要是为了进行数据处理时,则适宜建立成字节文件。若主要是为了输出到显示器或打印机供人们阅读,或者是为了供其他软件使用时,则适宜建立成字符文件。另外,当向字符文件输出一个换行符 `'\n'` 时,则将被看做为输出了回车 `'\r'` 和换行 `'\n'` 两个字符;相反,当从字符文件中读取回车和换行两个连续字符时,也被看做为一个换行符读取。

C++ 程序文件,利用其他各种语言编写的程序文件,用户建立的各种文本文件,各种软件系统中的帮助文件等,因都是 ASCII 码文件,所以都可以在 C++ 中作为字符文件使用。

C++ 系统把各种外部设备也看做为相应的文件。如把标准输入设备键盘和标准输出设备显示器看做为标准输入输出文件,其文件名(又称设备名)为 `con`。当向它输出信息时就是输出到显示器,当从它输入信息时就是从键盘输入。标准输入输出文件 `con` 对应两个系统预定义的流,即标准输入流 `cin` 和标准输出流 `cout`,分别用于键盘输入和显示器输出。由于键盘和显示器都属于字符设备,所以它们都是字符格式文件。以后对字符文件所介绍的访问操作也同样适用于键盘和显示器,而以前介绍的对键盘(`cin`)和显示器(`cout`)的访问操作也同样适用于所有字符文件。

无论是字符文件还是字节文件,在访问它之前都要定义一个文件流类的对象,并用该对象打开它,以后对该对象的访问操作就是对被它打开文件的访问操作。对文件操作结束后,再用该对象关闭它。对文件的访问操作包括输入和输出两种操作,输入操作是指从外部文件向内存变量输入数据,实际上是系统先把文件内容读入到该文件的内存缓冲区中,然后再从内存缓冲区中取出数据并赋给相应的内存变量,用于输入操作的文件称为输入文件。对文件的输出操作是指把内存变量或表达式的值写入到外部文件中,实际上是先写入到该文件的内存缓冲区中,待缓冲区被写满后,再由系统一次写入到外部文件中,用于输出操作的文件称为输出文件。

一个文件中保存的内容是按字节从数值 0 开始顺序编址的,文件开始位置的字节地址

为 0, 文件内容的最后一个字节的地址为  $n-1$  (假定文件长度为  $n$ , 即文件中所包含的字节数), 文件最后存放的文件结束符的地址为  $n$ , 它也是该文件的长度值。当一个文件为空时, 其开始位置和最后位置 (即文件结束符位置) 同为 0 地址位置。

对于每个打开的文件, 都存在着一个文件指针, 初始指向一个隐含的位置, 该位置由具体打开方式决定。每次对文件写入或读出信息都是从当前文件指针所指的位置开始的, 当写入或读出若干字节后, 文件指针就后移相应多个字节。当文件指针移动到最后, 读出的是文件结束符时, 则将使流对象调用 `eof()` 成员函数返回非 0 值 (通常为 1), 当然读出的是文件内容时将返回 0。文件结束符占有一个字节, 其值为 -1, 在 `ios` 类中把 EOF 常量定义为 -1。若利用字符变量依次读取字符文件中的每个字符, 当读取到的字符等于文件结束符 EOF 时则表示文件访问结束。

要在程序中使用文件时, 首先要在开始包含 `#include <fstream.h>` 命令。由它提供的输入文件流类 `ifstream`、输出文件流类 `ofstream` 和输入输出文件流类 `fstream` 定义用户所需要的文件流对象, 然后利用该对象调用相应类中的 `open` 成员函数, 按照一定的打开方式打开一个文件。文件被打开后, 就可以通过流对象访问它了, 访问结束后再通过流对象关闭它。

每个文件流类都有一个 `open` 成员函数, 并且具有完全相同的声明格式, 具体声明格式为:

```
void open(const char * fname, int mode);
```

其中 `fname` 参数用于指向要打开文件的文件名字符串, 该字符串内可以带有盘符和路径名, 若省略盘符和路径名则隐含为当前盘和当前路径。`mode` 参数用于指定打开文件的方式, 对应的实参是 `ios` 类中定义的 `open_mode` 枚举类型中的枚举常量, 或由这些枚举常量构成的按位或表达式。

`open_mode` 枚举类型中的每个枚举常量的含义如下:

```
ios::in           // 使文件只用于数据输入, 即从中读取数据。
ios::out          // 使文件只用于数据输出, 即向它写入数据。
ios::ate          // 使文件指针移至文件尾, 即最后位置。
ios::app          // 使文件指针移至文件尾, 并只允许向文件尾输出 (即追加) 数据。
ios::trunc        // 若打开的文件存在, 则清除其全部内容, 使之变为空文件。
ios::nocreate     // 若打开的文件不存在则不建立它, 返回打开失败信息。
ios::noreplace    // 若打开的文件存在, 则返回打开失败信息。
ios::binary       // 规定打开的为二进制文件, 否则打开的为字符文件。
```

下面对文件的打开方式作几点说明:

(1) 文件的打开方式可以为上述的一个枚举常量, 也可以为多个枚举常量构成的按位或表达式。如:

```
ios::in | ios::nocreate // 规定打开的文件是输入文件, 若文件不存在则返回
                          // 打开失败信息。
ios::in | ios::out      // 规定打开的文件同时用于输入和输出。
ios::app | ios::nocreate // 规定只向打开的文件尾追加数据, 若文件不存在
                          // 则返回打开失败信息。
ios::out | ios::noreplace // 规定打开的文件是输出文件, 若文件存在则返回
```



// 打开失败信息。

`ios::in | ios::out | ios::binary` // 规定打开的文件是二进制文件,并可同时  
// 用于输入和输出。

(2) 使用 `open` 成员函数打开一个文件时,若由字符指针参数所指定的文件不存在,则就建立该文件,当然建立的新文件是一个长度为 0 的空文件,但若打开方式参数中含有 `ios::nocreate` 选项,则不建立新文件,并且返回打开失败信息。

(3) 当打开方式中不含有 `ios::ate` 或 `ios::app` 选项时,则文件指针被自动移到文件的开始位置,即字节地址为 0 的位置。当打开方式中含有 `ios::out` 选项,但不含有 `ios::in`, `ios::ate` 或 `ios::app` 选项时,若打开的文件存在,则原有内容被清除,使之变为一个空文件。

(4) 当用输入文件流对象调用 `open` 成员函数打开一个文件时,打开方式参数可以省略,默认按 `ios::in` 方式打开,若打开方式参数中不含有 `ios::in` 选项时,则会自动被加上。当用输出文件流对象调用 `open` 成员函数打开一个文件时,打开方式参数也可以省略,默认按 `ios::out` 方式打开,若打开方式参数中不含有 `ios::out` 选项时,则也会自动被加上。

下面给出定义文件流对象和打开文件的一些例子:

(1) `ofstream fout;`

`fout.open("a: \\ xk.dat");` // 字符串中的双反斜线表示一个反斜线

(2) `ifstream fin;`

`fin.open("a: \\ wr.dat", ios::in | ios::nocreate);`

(3) `ofstream ofs;`

`ofs.open("a: \\ xk.dat", ios::app);`

(4) `fstream fio;`

`fio.open("a: \\ abc.ran", ios::in | ios::out | ios::binary);`

例子(1)首先定义了一个输出文件流对象 `fout`,使系统为其分配一个文件缓冲区,然后调用 `open` 成员函数打开 `a` 盘上的 `xk.dat` 文件,由于调用的成员函数省略了打开方式参数,所以采用默认的 `ios::out` 方式。执行这个调用时,若 `a:xk.dat` 文件存在,则清除该文件内容,使之成为一个空文件,若该文件不存在,则就在 `a` 盘上建立名为 `xk.dat` 的空文件。通过 `fout` 流打开 `a:xk.dat` 文件后,以后对 `fout` 流的输出操作就是对 `a:xk.dat` 文件的输出操作。

例子(2)首先定义了一个输入文件流对象 `fin`,并使其在内存中得到一个文件缓冲区,然后打开 `a` 盘上的 `wr.dat` 文件,并规定以输入方式进行访问,若该文件不存在则不建立新文件,使打开该文件的操作失败,此时由 `fin` 带回 0 值,由 `(! fin)` 是否为真判断文件是否打开。

例子(3)首先定义了一个输出文件流对象 `ofs`,同样在内存中得到一个文件缓冲区,然后打开 `a` 盘上已存在的 `xk.dat` 文件,并规定以追加数据的方式访问,即不破坏原有文件中的内容,只允许向尾部写入新的数据。

例子(4)首先定义了一个输入输出文件流对象 `fio`,同样在内存中得到一个文件缓冲区,然后按输入和输出方式打开 `a` 盘上的 `abc.ran` 二进制文件。此后既可以按字节向该文件写入信息,又可以从该文件读出信息。

在每一种文件流类中,既定义有无参构造函数,又定义有带参构造函数,并且所带参数与 `open` 成员函数所带参数完全相同。当定义一个带有实参表的文件流对象时,将自动调用相应的带参构造函数,打开第一个实参所指向的文件,并规定按第二个实参所给的打开方式

进行操作。所以它同先定义不带参数的文件流对象,后通过流对象调用 `open` 成员函数打开文件的功能完全相同。对于上述给出的四个例子,依次与下面的文件流定义语句功能相同。

- (1) `ofstream fout("a: \\ xk.dat");`
- (2) `ifstream fin("a: \\ wr.dat", ios::in | ios::nocreate);`
- (3) `ofstream ofs("a: \\ xk.dat", ios::app);`
- (4) `fstream ofs(a: \\ abc.ran", ios::in | ios::out | ios::binary);`

每个文件流类中都提供有一个关闭文件的成员函数 `close()`,当打开的文件操作结束后,就需要关闭它,使文件流与对应的物理文件断开联系,并能够保证最后输出到文件缓冲区中的内容,无论是否已满,都将立即写入到对应的物理文件中。文件流对应的文件被关闭后,还可以利用该文件流调用 `open` 成员函数打开其他的文件。

关闭任何一个流对象所对应的文件,就是用这个流对象调用 `close()` 成员函数即可。如要关闭 `fout` 流所对应的 `a: \\ xk.dat` 文件,则关闭语句为:

```
fout.close();
```

### 10.3.2 字符文件的访问操作

C++ 文件包括字符文件和字节文件两种类型,对它们的访问操作各不相同。这一小节专门讨论对字符文件的访问操作,下一小节再讨论对字节文件的访问操作。

当只需要对数据进行顺序输入输出操作时,则适合使用字符文件。对字符文件的访问操作包括向字符文件顺序输出数据和从字符文件顺序输入数据这两个方面。所谓顺序输出就是依次把数据写入到文件的末尾(当然文件结束符也随之后移,它始终占据整个文件空间的最后一个字节位置),顺序输入就是从文件开始位置起依次向后提取数据,直到碰到文件结束符为止。

#### 1. 向字符文件输出数据

向字符文件输出数据有两种方法,一种是调用从 `ostream` 流类中继承来的插入操作符重载函数,另一种是调用从 `ostream` 流类中继承来的 `put` 成员函数。它们的声明格式如下:

```
ostream& operator << (简单类型);  
ostream& put( char );
```

采用第一种方法时,插入操作符左边是文件流对象,右边是要输出到该文件流(即对应的文件)中数据项。当系统执行这种插入操作时,首先计算出插入操作符右边数据项(即表达式)的值,接着根据该值的类型调用相应的插入操作符重载函数,把这个值插入(即输出)到插入操作符左边的文件流中,然后返回这个流,以便在一条输出语句中继续输出其他数据。

若要向字符文件中插入一个用户定义类型的数据,除了可以将每个域的值依次插入外,还可以进行整体插入。对于后者,要预先定义有对该类型数据进行插入操作符重载的函数。

采用第二种方法时,文件流对象通过点操作符、文件流指针通过箭头操作符调用成员函数 `put`。当执行这种调用操作时,首先向文件流中输出一个字符,即实参的值,然后返回这个

文件流。

下面给出几个进行字符文件操作的例子。

例 1. 向 a 盘上的 wr1.dat 文件输出 0~20 之间的整数,含 0 和 20 在内。

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
void main(void)
{
    ofstream f1("a:wr1.dat");
    // 定义输出文件流,并打开相应文件,若打开失败则 f1 带回 0 值
    if (!f1) { // 当 f1 打开失败时进行错误处理
        cerr << "a:wr1.dat file not open!" << endl;
        exit(1);
    }
    for(int i=0; i<21; i++)
        f1 << i << " "; // 向 f1 文件流输出 i 值
    f1.close(); // 关闭 f1 所对应的文件
}
```

例 2. 把从键盘上输入的若干行文本字符原原本本地存入到 a 盘上 wr2.dat 文件中,直到从键盘上按下 Ctrl+z 组合键为止。此组合键代表文件结束符 EOF。

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
void main(void)
{
    char ch;
    ofstream f2("a:wr2.dat");
    if (!f2) { // 当 f1 打开失败时进行错误处理
        cerr << "File of a:wr2.dat not open!" << endl;
        exit(1);
    }
    ch = cin.get(); // 从 cin 流中提取一个字符到 ch 中
    while(ch != EOF) {
        f2.put(ch); // 把 ch 字符写入到 f2 流中,此语句也
        // 可用 f2 << ch 代替
        ch = cin.get(); // 从 cin 流中提取下一个字符到 ch 中
    }
    f2.close(); // 关闭 f2 所对应的文件
}
```

例 3. 假定一个结构数组 a 中的元素类型 pupil 包含有表示姓名的字符指针域 name 和表示成绩的整数域 grade,试编写一个函数把该数组中的 n 个元素输出到字符文件“a:wr3.dat”中。

```
#include <stdlib.h>
#include <fstream.h>
void ArrayOut(pupil a[], int n)
{
```

```

ofstream f3("a:wr3.dat");
if (!f3) { // 当 f3 打开失败时进行错误处理
    cerr << "File of a:wr3.dat not open!" << endl;
    exit(1);
}
for(int i=0; i<n; i++)
    f3 << a[i].name << endl << a[i].grade << endl;
f3.close();
}

```

若已经为输出 pupil 类型的数据定义了如下插入操作符重载函数:

```

ostream& operator << (ostream& ostr, pupil& x)
{
    ostr << x.name << endl << x.grade << endl;
    return ostr;
}

```

则可将上述函数中 for 循环体语句修改为“f3 << a[i];”。

## 2. 从字符文件输入数据

从打开的字符文件中输入数据到内存变量有三种方法。一种是调用提取操作符重载成员函数,每次从文件流中提取用空白符隔开(当然最后一个数据以文件结束符为结束标志)的一个数据,这同使用提取操作符从 cin 流中读取数据的过程和规定完全相同,在读取一个数据前文件指针自动跳过空白字符,向后移到非空白字符时读取一个数据。第二种是调用 get() 成员函数,每次从文件流中提取一个字符(不跳过任何字符,当然回车和换行两个字符被作为一个换行字符看待)并作为返回值返回,或者调用 get(char&) 成员函数,每次从文件流中提取一个字符到引用变量中,同样不跳过任何字符。第三种是调用 getline(char \* buffer, int len, char = '\n') 成员函数,每次从文件流中提取以换行符隔开(当然最后一行数据以文件结束符为结束标志)的一行字符到字符指针 buffer 所指向的存储空间中,若碰到换行符之前所提取字符的个数大于等于参数 len 的值,则本次只提取 len - 1 个字符,被提取的一行字符是作为字符串写入到 buffer 所指向的存储空间中的,也就是说在一行字符的最后位置必须写入 '\0' 字符。文件流调用的上述各种成员函数都是在 istream 流类中定义的,它们都被每一种文件流类继承了下来,所以文件流类的对象可以直接调用它们。由于 cin 和 cout 流对象所属的流类也分别是 istream 流类和 ostream 流类的派生类,所以 cin 和 cout 也可以直接调用相应流类中的成员函数。

上述介绍的在 istream 流类中的每个成员函数的声明格式分别如下:

```

istream& operator >> (简单类型 &);
// 从流中提取一个数据到引用对象中
int get(); // 返回从流中提取到的一个字符
istream& get(char&); // 从流中提取一个字符到字符引用中
istream& getline(char * buffer, int len, char = '\n');
// 从流中提取一行字符到由字符指针所指向的存储空间中

```

当使用流对象调用 get() 成员函数时,通过判断返回值是否等于文件结束符 EOF 可知文

件中的数据是否被输入完毕。当使用流对象调用其他三个成员函数时,若提取成功则返回非 0 值,若提取失败(即已经读到文件结束符,未读到文件内容)则返回 0 值。

在通常情况下,若一个文件是使用插入操作符输出数据而建立的,则当做输入文件打开后,应使用提取操作符输入数据;若一个文件是使用 put 成员函数输出字符而建立的,则当做输入文件打开后,应使用 get()或 get(char&)成员函数输入字符数据;若每次需要从一个输入文件中读入一行字符时,则需要使用 getline 成员函数。

下面给出进行字符文件输入操作的几个例子。

例 4. 从本小节例 1 所建立的 a:wr1.dat 文件中输入全部数据并依次显示到屏幕上。

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
void main(void)
{
    ifstream f1("a:wr1.dat", ios::in | ios::nocreate);
    // 定义输入文件流,并打开相应文件,若打开失败则 f1 带回 0 值
    if (!f1) { // 当 f1 打开失败时进行错误处理
        cerr << "a:wr1.dat file not open!" << endl;
        exit(1);
    }
    int x;
    while(f1 >> x) // 依次从文件中输入整数到 x,
        // 当读到的是文件结束符时条件表达式的值为 0
        cout << x << ' ';
    cout << endl;
    f1.close(); // 关闭 f1 所对应的文件
}
```

该程序运行结果如下:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

例 5. 从本节例 2 所建立的 a:wr2.dat 文件中按字符输入全部数据,把它们依次显示到屏幕上,并且统计出文件内容中的行数。

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
void main(void)
{
    ifstream f2("a:wr2.dat", ios::in | ios::nocreate);
    if (!f2) {
        cerr << "a:wr2.dat file not open!" << endl;
        exit(1);
    }
    char ch; // 用 ch 读入字符
    int i = 0; // 用 i 统计行数
    while(f2.get(ch)) // 依次从文件中输入字符到 ch,当读到的是文件
        // 结束符时条件表达式的值为 0。
    {
```

```

        cout << ch;
        if(ch == '\n') i++;
    }
    cout << endl << "lines: " << i << endl;
    f2.close(); // 关闭 f1 所对应的文件
}

```

若把 while 语句中的条件表达式(f2.get(ch))换为((ch = f2.get()) != EOF)完全相同。  
该程序运行后的显示结果如下：

```

12345
asdfghj ghjk
wqregwrew

lines: 3

```

其中前三行字符就是建立文件时从键盘上输入的文本,在此被原原本本地显示出来。

例 6. 在屏幕上显示出本小节例 3 所建立的 a:wr3.dat 文件中具有最高成绩的学生记录,同时统计出所存的学生记录数。

```

#include<iostream.h>
#include<stdlib.h>
#include<fstream.h>
struct pupil {
    char name[10];
    int grade;
};
void main(void)
{
    ifstream f3("a:wr3.dat", ios::in | ios::nocreate);
    if (!f3) {
        cerr << "a:wr3.dat file not open!" << endl;
        exit(1);
    }
    pupil x, w = {"", 0};
    // 用 x 读取记录,用 w 保存当前得到的成绩最高的记录
    int n = 0; // 用 n 统计学生记录数,
    while(f3 >> x.name) {
        f3 >> x.grade;
        n++;
        if(x.grade > w.grade) w = x;
    }
    cout << "记录个数: " << n << endl;
    cout << "成绩最高的学生: " << w.name << " " << w.grade << endl;
    f3.close();
}

```

假定 a:wr3.dat 文件中的内容如下：

```

xxk 89
wr 46
xc 68

```

xxh 25

则程序运行结果如下:

记录个数: 4

成绩最高的学生: xxk 89

例 7. 首先从键盘上输入若干行字符到一个二维字符数组 **a** 中, 然后分别统计出 **a** 中保存的字母、数字和其他字符的个数。

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
void main()
{
    int n;
    cout << "待输入文本的行数: ";
    cin >> n >> ws; // 用操纵符 ws 读取输入 n 值后的换行符
    char (* a)[80] = new char[n][80];
    // 动态分配具有 n 行 80 列的二维字符数组空间
    int i;
    for(i = 0; i < n; i++) { // 从键盘上输入 n 行文本到数组 a 中
        cin.getline(a[i], 80);
    }
    int c1, c2, c3;
    // 用它们分别统计数组 a 中字母、数字和其他字符的个数
    c1 = c2 = c3 = 0;
    for(i = 0; i < n; i++) {
        char * p = a[i]; // 将一行字符的首地址赋给字符指针 p
        char ch = * p; // 将该行首字符赋给 ch
        while(ch) { // 当 ch 不为字符串结尾的空字符时则进入循环
            if(ch >= 65 && ch <= 90 || ch >= 97 && ch <= 122)
                c1++; // 统计字母个数
            else if(ch >= 48 && ch <= 57)
                c2++; // 统计数字个数
            else
                c3++; // 统计其他字符个数
            ch = * ++p; // 得到该行的下一个字符
        }
    }
    cout << endl;
    cout << "c1 = " << c1 << ", c2 = " << c2 << ", c3 = " << c3 << endl;
}
```

该程序输入和运行结果如下:

待输入文本的行数: 3

123 456

qweASD,.;{[])(\* & +;

zxzAQ!@ # \$ %

c1 = 10, c2 = 6, c3 = 19

在 `istream` 流类中还有两个成员函数 `peek()` 和 `putback(char)` 供输入文件流对象和 `cin` 对象调用。当调用 `peek()` 成员函数时返回一个整数值,它是从流中提取一个字符的 ASCII 码,但利用它提取字符后,文件指针不向后移动,而是仍停留在原有位置上。当调用 `putback(char)` 成员函数时,将把一个字符(即值参的值,它通常为刚读到的字符)重新放回到原有位置上,即当前文件指针所指的前一个位置上,使得输入流状态恢复为最近一次提取字符前的状态。在下面的例 8 中给出了调用 `putback(char)` 成员函数的情况。

例 8. 假定在 a 盘上建立的 `xk1.txt` 文件中存放着字符串和浮点数的序列,现要求把所有字符串依次保存到 `a:xk2.txt` 文件中,把所有浮点数依次保存到 `a:xk3.txt` 文件中。

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
void main(void)
{
    ifstream f1("a:xk1.txt", ios::in | ios::nocreate);
    ofstream f2("a:xk2.txt");
    ofstream f3("a:xk3.txt");
    char ch;
    char s[20];
    float x;
    while(f1.get(ch)) {
        if(ch >= 65 && ch <= 90 || ch >= 97 && ch <= 122)
        { // 从 f1 中读入一个字符串到 s 中,然后再把它写入 f2 中
            f1.putback(ch); // 向流中压回刚读到的字母
            f1 >> s;
            f2 << s << ' ';
        }
        else if(ch >= 48 && ch <= 57 || ch == 46)
        { // 从 f1 中读入一个浮点数到 x 中,然后再把它写入 f3 中
            f1.putback(ch); // 向流中压回刚读到的数字或小数点
            f1 >> x;
            f3 << x << ' ';
        }
    }
    f1.close(); f2.close(); f3.close();
}
```

若要依次输出 `a:xk2.txt` 文件中保存的所有字符串,则可以用此文件名作为实参调用如下的函数:

```
void print(char * fname)
{
    ifstream fin(fname);
    char a[20];
    while(fin >> a)
        cout << a << endl;
    fin.close();
}
```



若要依次输出 a:rk3.txt 文件中保存的所有浮点数,则可以用此文件名作为实参调用上面函数,但上面函数中的“char a[20];”语句要修改为“float a;”语句。

### 10.3.3 字节文件的访问操作

字节文件是指在打开方式中带有 ios::binary 选项的文件。字节文件可以是输入文件、输出文件、既输入又输出的文件。向字节文件中输出信息时,就是把内存中由指定字符指针所指向的具有一定字节数的内容原原本本地写入到文件中。当然所写内容是从当前文件指针所指位置开始向后存放,然后文件指针被自动后移所写入内容的字节数。从字节文件中输入信息就是把具有一定字节数的内容原原本本地拷贝到内存中由指定字符指针所指向的存储空间中。当然输入信息是从当前文件指针所指位置开始读出,然后文件指针被自动后移所读出内容的字节数。

一个文件被用户定义的一个文件流对象打开后,通过文件流对象调用在 istream 流类中定义的 read 成员函数能够从文件流对象所对应的文件中读出信息,通过文件流对象调用在 ostream 流类中定义的 write 成员函数能够向文件流对象所对应的文件中写入信息。这两个成员函数的声明格式如下:

```
istream& read(char* buffer, int len);  
ostream& write(const char* buffer, int len);
```

其中字符指针 buffer 用于存放内存中保存文件读写信息的一块存储空间的首地址,整型参数 len 用于存放一次读写文件的字节数。当调用 read 成员函数时若读到了 len 个字节内容,则返回非 0 值,若读到了文件结束符,则返回 0 值,此时通过调用 istream 流类中提供的 gcount() 成员函数能够返回实际读取的字节数。

在每个文件中都存在着一个文件指针,利用 istream 流类中提供的 seekg 成员函数能够把输入文件中的文件指针移动到指定的位置上;利用 ostream 流类中提供的 seekp 成员函数能够把输出文件中的文件指针移动到指定的位置上;若一个文件既按输入又按输出打开,则既可以使用 seekg 又可以使用 seekp 移动文件指针。这两个成员函数的声明格式如下:

```
istream& seekg(long dis, seek_dir ref = ios::beg);  
ostream& seekp(long dis, seek_dir ref = ios::beg);
```

其中 seek\_dir 是一个在 ios 根基类中定义的枚举类型,它包含有三个常量:ios::beg, ios::cur 和 ios::end。函数中的 ref 参数取这三个常量值之一,其中 ios::beg 常量是它的默认值。ref 参数的作用是指定移动文件指针的参考点,当分别取 ios::beg, ios::cur 或 ios::end 时,则参考点分别为文件开始位置(字节地址为 0 的位置)、当前文件指针位置和文件结尾位置(即文件结束符位置)。当利用一个文件流对象调用上述一个函数后,文件指针被移到距离参考点 ref 的 abs(dis) 个字节(abs 为取绝对值的函数,在 math.h 中头文件中声明)的位置上,当 dis 为正时则表示后移,为负时则表示前移。因此当 ref 被指定为 ios::beg 时,dis 不能小于 0;当 ref 被指定为 ios::end 时,dis 不能大于 0;当 ref 被指定为 ios::cur 时,dis 既可以大于等于 0,也可以小于等于 0。由于这两个成员函数的第二个参数都带有默认值,所以调用它们时可以只使用一个实参,此时将使文件指针移到字节地址为实参值的位置上。

在 `istream` 流类和 `ostream` 流类中还分别定义有 `tellg()` 和 `tellp()` 成员函数供文件流对象调用,用来分别返回输入文件和输出文件中文件指针的位置,即对应的字节地址。对于既按输入又按输出方式打开的文件,为了返回当前文件指针位置,既可以调用 `tellg()`,也可以调用 `tellp()` 实现。

上述讨论的按字节读写文件、移动文件指针和得到文件指针位置的操作,不仅适应于字节文件,而且也适应于字符文件,但主要应用于字节文件中,较少应用于字符文件中。

下面给出进行字节文件操作的几个例子。

例1. 首先利用 48,62,25,73,66,80,78,54,36,47 等 10 个整数初始化一个整型数组 `a[10]`,然后把 `a` 中每个元素的值依次写入到字节文件 `a:shf1.dat` 中。

```
#include <stdlib.h>
#include <fstream.h>
void main(void)
{
    ofstream f1("a:shf1.dat", ios::out | ios::binary);
    // 定义输出文件流,并打开相应的字节文件
    if (!f1) {
        cerr << "a:shf1.dat file not open!" << endl;
        exit(1);
    }
    int a[10] = {48,62,25,73,66,80,78,54,36,47};
    for(int i=0;i<10;i++) // 向 f1 对应的文件中输出每个元素值
        f1.write((char*)&a[i], sizeof(int));
    f1.close(); // 关闭 f1 所对应的文件
}
```

例2. 求出 `a:shf1.dat` 文件中保存的所有整数的最大值、最小值和平均值。

```
#include <stdlib.h>
#include <fstream.h>
void main(void)
{
    ifstream f1("a:shf1.dat", ios::in | ios::nocreate | ios::binary);
    // 定义输入文件流,并打开相应的字节文件
    if (!f1) {
        cerr << "a:shf1.dat file not open!" << endl;
        exit(1);
    }
    int x,max,min;
    float mean;
    f1.seekg(0, ios::end); // 将文件指针移至文件尾,
    // 此时文件指针位置就是按字节计算的文件长度。
    int n = f1.tellg()/sizeof(int);
    // n 值为按整型大小计算的文件长度
    if(n == 0) {cerr << "输入文件为空!" << endl; exit(1);}
    // 若文件为空则返回
    f1.seekg(0);
    // 将文件指针移至文件开始位置
    f1.read((char*)&x, sizeof(int));
```

```

        // 读取第一个整数到 x 中
max = min = x;
        // 给保存最大值和最小值的变量赋初值
mean = (float)x/n;
        // 给保存平均值的变量赋初值
for(int i=1; i <= n-1; i++)
{ // 依次读取 f1 对应文件中的 n-1 个整数
    fl.read((char*)&x, sizeof(int));
    if(x > max)
        max = x; // 较大的值放入 max 中
    else if(x < min)
        min = x; // 较小的值放入 min 中
    mean += (float)x/n;
        // 把 x 对应的平均值累加到 mean 中
}
cout << "maximum: " << max << endl; // 输出最大值
cout << "minimum: " << min << endl; // 输出最小值
cout << "mean: " << mean << endl; // 输出平均值
fl.close();
}

```

程序中的 11~36 行也可编写为如下程序段:

```

int x,max,min,n=0;
float mean;
if(fl.read((char*)&x, sizeof(int)))
    n++;
else
    (cerr << "输入文件为空!" << endl; exit(1);)
    // 若文件为空则返回
max = min = x; mean = float(x);
while(!fl.eof()) {
    if(fl.read((char*)&x, sizeof(int))) {
        n++;
        if(x > max)
            max = x;
        else if(x < min)
            min = x;
        mean += x;
    }
}
mean /= n;

```

该程序运行结果如下:

```

maximum: 80
minimum: 25
mean: 56.9

```

例 3. 从键盘上输入若干条 pupil 类型的学生记录到 a:shf2.dat 字节文件中,当按下 Ctrl + z 组合键(即输入文件结束符)后终止输入。

```

#include <stdlib.h>
#include <fstream.h>
struct pupil {
    char name[10];
    int grade;
};
void main(void)
{
    fstream fout("a:shf2.dat", ios::out | ios::trunc | ios::binary);
    if (!fout) {
        cerr << "a:shf2.dat file not open!" << endl;
        exit(1);
    }
    pupil x;
    cout << "请输入若干条学生记录,按 Ctrl + z 键后结束:" << endl;
    while(cin >> x.name) {
        cin >> x.grade;
        fout.write((char *)&x, sizeof(x));
    }
    fout.close();
    cout << "输入结束." << endl;
}

```

假定从键盘上输入以下记录,它们都被依次存入到 a:shf2.dat 文件中了。

```

zhshj 76
hgyin 84
shian 68
zhb 92
zjmin 70
xjip 63

```

例 4. 编一程序,对上一例题中建立的 a:shf2.dat 文件实现如下操作功能:

- (1) 向文件尾追加一条记录;
- (2) 从文件中查找给定姓名的记录,若查找成功则返回 true 并由引用参数带回该记录,否则返回 false 表示查找失败;
- (3) 更新(即修改)给定姓名的记录为新输入的记录,若更新成功则返回 true,否则返回 false;
- (4) 向屏幕打印输出文件中的所有记录;
- (5) 结束运行。

下面给出实现此题功能的参考程序,请读者自行阅读和分析。

```

#include <stdlib.h>
#include <string.h>
#include <fstream.h>
struct pupil {
    char name[10];
    int grade;
};

```

```

void Append(fstream& fio, int& n, const pupil& rec)
{
    fio.seekp(0, ios::end);
    fio.write((char*)&rec, sizeof(rec));
    n++;
}

bool Find(fstream& fio, int n, const pupil& rec)
{
    fio.seekg(0);
    pupil x;
    for(int i=0; i<n; i++) {
        if(fio.read((char*)&x, sizeof(x)))
            if(strcmp(x.name, rec.name) == 0) {
                cout << "记录被找到! " << x.name << ' ' << x.grade << endl;
                rec = x; // 由引用参数 rec 带回被查找到的记录
                return true;
            }
    }
    cout << "没有查找到姓名为 " << rec.name << " 的记录 ." << endl;
    return false;
}

bool Update(fstream& fio, int n, const pupil& rec)
{
    fio.seekg(0);
    pupil x;
    int m = sizeof(x);
    for(int i=0; i<n; i++) {
        if(fio.read((char*)&x, m))
            if(strcmp(x.name, rec.name) == 0) {
                fio.seekg(-m, ios::cur);
                fio.write((char*)&rec, m);
                cout << rec.name << " 的记录被更新!" << endl;
                return true;
            }
    }
    cout << "没有查找到姓名为 " << rec.name << " 的待更新的记录 ." << endl;
    return false;
}

void Print(fstream& fio, int n)
{
    fio.seekg(0);
    pupil x;
    for(int i=0; i<n; i++) {
        fio.read((char*)&x, sizeof(x));
        cout << x.name << ' ' << x.grade << endl;
    }
}

```

```

void main(void)
{
    fstream ff("a:shf2.dat", ios::in | ios::out |
               ios::nocreate | ios::binary);
    if (!ff) {
        cerr << "a:shf2.dat file not open!" << endl;
        exit(1);
    }
    pupil x;
    int i;
    ff.seekg(0, ios::end);
    int n = ff.tellg() / sizeof(x);
    while(1) {
        cout << "功能号表: " << endl << endl;
        cout << "1 --- 向文件追加一条记录;" << endl;
        cout << "2 --- 按姓名查找记录;" << endl;
        cout << "3 --- 按姓名更新记录;" << endl;
        cout << "4 --- 向屏幕输出文件中的所有记录;" << endl;
        cout << "5 --- 结束运行 ." << endl << endl;
        cout << "请输入您的选择(1~5): ";
        cin >> i;
        switch (i) {
            case 1:
                cout << "输入待追加学生的记录: ";
                cin >> x.name >> x.grade;
                Append(ff, n, x);
                break;
            case 2:
                cout << "输入待查找学生的姓名: ";
                cin >> x.name;
                Find(ff, n, x);
                break;
            case 3:
                cout << "输入待更新学生的记录: ";
                cin >> x.name >> x.grade;
                Update(ff, n, x);
                break;
            case 4:
                cout << "a:shf2.dat 文件中的全部记录:" << endl;
                Print(ff, n);
                break;
            case 5:
                cout << endl << "结束运行, 再见!" << endl;
                return;
        }
    }
    ff.close();
}

```

假定把姓名为 xjip 的记录更新为 |xjip, 88|, 并把 |wping, 77| 的记录追加到文件中, 则选择功能表中的 4 后, 则屏幕上显示出的文件内容为:

```
请输入您的选择(1~5): 4
a:shf2.dat 文件中的全部记录:
zhshj 76
hgyin 84
shian 68
zhib 92
zjmin 70
xjip 88
wping 77
```

## 10.4 字符串流

字符串流类包括输入字符串流类 `istream`, 输出字符串流类 `ostream` 和输入输出字符串流类 `stringstream` 三种。它们都被定义在系统头文件 `strstream.h` 中。只要在程序中带有该头文件, 就可以使用任一种字符串流类定义字符串流对象。每个字符串流对象简称为字符串流。

字符串流对应的访问空间是内存中由用户定义的字符数组, 而文件流对应的访问空间是外存上由文件名确定的文件存储空间。由于字符串流和文件流都是输入流类 `istream` 和输出流类 `ostream` 的继承类, 所以对它们的操作方法基本相同。但仍有一点区别: 就是每个文件都有文件结束符标志, 利用它可以判断读取数据是否到达文件尾, 而字符串流所对应的字符数组中没有相应的结束符标志, 这只能靠用户规定一个特殊字符作为其结束符使用, 在向字符串流对应的字符数组写入所有数据后, 再写入它表示结束。

每一种字符串流类都不带有 `open` 成员函数, 所以只有在定义字符串流的同时给出必要的参数, 通过自动调用相应的构造函数来使之与一个字符数组发生联系。以后对字符串流的操作实质上就是在该数组上进行的, 就像对文件流的操作实质上就是在对应文件上进行的情况一样。三种字符串流类的构造函数声明格式分别如下:

```
istream(const char * buffer);
ostream(char * buffer, int n);
stringstream(char * buffer, int n, int mode);
```

调用第一种构造函数建立的是输入字符串流, 对应的字符数组空间由 `buffer` 指针所指向。调用第二种构造函数建立的是输出字符串流, 对应的字符数组空间同样由 `buffer` 指针所指向, 该空间的大小(即字节数)由参数表中第二个参数给出。调用第三种构造函数建立的是输入输出字符串流, 其中第一个参数指定对应的字符数组存储空间, 第二个参数指定空间的大小, 第三个参数指定打开方式。一个字符串流被定义后就可以调用相应的成员函数进行数据的输入、输出操作, 就如同使用文件流调用相应的成员函数进行有关操作一样。下面给出定义相应字符串流的例子。

- (1) `ostream sout(a1, 50);`
- (2) `istream sin(a2);`
- (3) `stringstream sio(a3, sizeof(a3), ios::in | ios::out);`

第(1)条语句定义了一个输出字符串流 `sout`, 使用的字符数组为 `a1`, 大小为 50 个字节, 以后对 `sout` 的输出都将被写入到字符数组 `a1` 中。第(2)条语句定义了一个输入字符串流

sin,使用的字符数组为 a2,以后从 sin 中读取的输入数据都将来自字符数组 a2 中。第(3)条语句定义了一个输入输出字符串流 sio,使用的字符数组为 a3,大小为 a3 数组的长度,打开方式规定为既能够用于输入又能够用于输出,当然进行输入的数据来自数组 a3,进行输出的数据写入数组 a3。

对字符串流的操作方法通常与对字符文件流的操作方法相同。下面给出一些使用字符串流的例子。

例 1. 从一个字符串流中输入用逗号分开的每一个整数并显示出来。

程序如下:

```
#include <strstream.h>
void main()
{
    char a[] = "38,46,55,78,42,77,60,93@";
    cout << a << endl; // 输出 a 字符串
    istrstream sin(a);
    // 定义一个输入字符串流 sin,使用的字符数组为 a.
    char ch = ' ';
    int x;
    while(ch != '@') { // 使用 '@' 字符作为字符串流结束标志
        sin >> ws >> x >> ws;
        // 从流中读入一个整数,并使用操作符 ws 读取
        // 一个整数前后的空白字符
        cout << x << ' ';
        // 输出 x 的值并后跟一个空格
        sin.get(ch);
        // 从 sin 流中读入一个字符,实际读取的是 ',' 或 '@' 字符
    }
    cout << endl;
}
```

该程序运行结果如下:

```
38,46,55,78,42,77,60,93@
38 46 55 78 42 77 60 93
```

例 2. 从一个字符串中得到每一个整数,并把它们依次存入到一个字符串流中,最后向屏幕输出这个字符串流。

分析:假定待处理的一个字符串是从键盘上输入得到的,把它存入到字符数组 a 中,并且要把 a 定义为一个输入字符串流 sin。还需要定义一个输出字符串流 sout,假定对应的字符数组为 b,用它保存依次从输入流中得到的整数。该程序的处理过程需要使用一个 while 循环,每次从 sin 流中得到一个整数,并把它输出到 sout 流中。然后要向 sout 流中输出一个作为字符串流结束符使用的特殊字符(假定为 '@')和字符串结束符 '\0'。最后向屏幕输出 sout 流所对应的字符串。整个程序如下:

```
#include <strstream.h>
void main()
{
    char a[50];
```



```

char b[50];
istream sin(a);
    // 定义一个输入字符串流 sin, 使用的字符数组为 a
ostream sout(b, sizeof(b));
    // 定义一个输出字符串流 sout, 使用的字符数组为 b
cin.getline(a, sizeof(a));
    // 假定从键盘上输入的字符串为:
    // "ab38+56,46*55-23/ad663,WER40ff:dy{63;44}@
char ch=' ';
int x;
while(ch!='@') { // 使用 '@' 字符作为字符串流结束标志
    if(ch >= 48 && ch <= 57) {
        sin.putback(ch); // 把刚读入的一个数字压回流中。
        sin >> x; // 从流中读入一个整数, 当碰到非数字字
            // 符时则就认为一个整数结束。
        sout << x << ' ';
            // 将 x 输出到字符串流 sout 中
    }
    sin.get(ch);
        // 从 sin 流中读入下一个字符
}
sout << '@' << ends; // 向 sout 流输出作为结束符的 '@' 字符和
    // 一个字符串结束符 '\0'
cout << b; // 输出字符串流 sout 对应的字符串
cout << endl;
}

```

该程序的运行结果如下:

```

ab38+56,46*55-23/ad663,WER40ff:dy{63;44}@
38 56 46 55 23 663 40 63 44 @

```

## 习题十

### (一) 单选题

1. 当使用 `ifstream` 流类定义一个流对象并打开一个磁盘文件时, 文件的隐含打开方式为\_\_\_\_\_。  
 A `ios::in`      B `ios::out`      C `ios::in | ios::out`      D `ios::binary`
2. 当使用 `ofstream` 流类定义一个流对象并打开一个磁盘文件时, 文件的隐含打开方式为\_\_\_\_\_。  
 A `ios::in`      B `ios::out`      C `ios::in | ios::out`      D `ios::binary`
3. 当使用 `fstream` 流类定义一个流对象并打开一个磁盘文件时, 文件的隐含打开方式为\_\_\_\_\_。  
 A `ios::in`      B `ios::out`      C `ios::in | ios::out`      D 没有
4. 当需要使用 `istream` 流类定义一个流对象并联系一个字符串时, 应在文件开始使用

#include 命令,使之包含\_\_\_\_\_文件。

A iostream.h    B iomanip.h    C fstream.h    D strstream.h

5. 当需要使用 ostream 流类定义一个流对象并联系一个字符串时,应在文件开始使用

#include 命令,使之包含\_\_\_\_\_文件。

A iostream.h    B stdlib.h    C strstream.h    D fstream.h

## (二) 填空题

1. 在C++ 流类库中,根基类为\_\_\_\_\_。

2. 在C++ 流类库中,输入流类和输出流类的名称分别为\_\_\_\_\_和\_\_\_\_\_。

3. 若要在程序文件中进行标准输入输出操作,则必须在开始的 #include 命令中使用\_\_\_\_\_头文件。

4. 若要在程序文件中进行文件输入输出操作,则必须在开始的 #include 命令中使用\_\_\_\_\_头文件。

5. 当从字符文件中读取回车和换行两个字符时,被系统看做为一个\_\_\_\_\_。

## (三) 给出下列每个程序或函数的执行结果

1. void xxx1()

```
{
    int x,y;
    x=20; y=70;
    cout <<"dec: " << dec << x << ' ' << y << endl;
    cout <<"oct: " << oct << x << ' ' << y << endl;
    cout <<"hex: " << hex << x << ' ' << y << endl;
    cout << dec;
}
```

2. #include <iomanip.h>

```
void main() {
    struct AB {
        char aa[15];
        int bb;
    };
    AB a[4] = {{"Apple",25}, {"Peach",40}, {"Pear",36}, {"Tomato",62}};
    cout.setf(ios::left);
    for(int i=0; i<4; i++) {
        cout << setw(10) << a[i].aa;
        cout << setw(10) << a[i].bb << endl;
    }
    cout << resetiosflags(ios::left);
}
```

3. void xxx2()

```
{
    double radius,area;
    radius=2.5;
    area=3.14159*radius*radius;
```

```

        cout << setw(10) << "radius = " << setw(10) << radius;
        cout << setw(10) << "area = " << setw(10) << area << endl;
        cout.setf(ios::left);
        cout << setw(10) << "radius = " << setw(10) << radius;
        cout << setw(10) << "area = " << setw(10) << area << endl;
        cout.unsetf(ios::left);
        cout << setw(10) << "radius = ";
        cout << setiosflags(ios::left) << setw(10) << radius;
        cout.unsetf(ios::left);
        cout << setw(10) << "area = ";
        cout << setiosflags(ios::left) << setw(10) << area << endl;
    }

4. void xok3()
{
    double radius, area;
    radius = 2.5;
    area = 3.14159 * radius * radius;
    cout << setw(10) << radius << setw(10) << area << endl;
    cout.setf(ios::showpoint);
    cout.precision(4);
    cout << setw(10) << radius << setw(10) << area << endl;
    cout.unsetf(ios::showpoint);
    cout << setprecision(5);
    cout << setw(10) << radius << setw(10) << area << endl;
}

5. void xok4()
{
    char a[] = "36 94 135 -1";
    istringstream str(a);
    int k;
    str >> k;
    while(k! = -1) {
        cout << dec << k << ' ' << oct << k << ' ' << hex << k << endl;
        str >> k;
    }
}

```

#### (四) 写出下列每个程序或函数的功能

```

1. #include <iomanip.h>
   #include <fstream.h>
   #include <string.h>
   void JA(char * fname)
       // 可把以 fname 所指字符串作为文件标识符的文件称为 fname 文件
   {
       ofstream fout(fname);
       char a[20];
       cin >> a;
       while(strcmp(a, "end") != 0) {

```

```

        fout << a << endl;
        cin >> a; *
    }
    fout.close();
}

2. void wr1(char * fname)
{
    ofstream file(fname);
    int x;
    cout << "input data:" << endl;
    while(1) {
        cin >> x;
        if(x != -1) file << x << ' ';
        else break;
    }
    file.close();
}

3. void wr2(char * fname)
{
    ifstream file(fname, ios::in | ios::nocreate);
    int x;
    while(file >> x)
        cout << x << ' ';
    cout << endl;
    file.close();
}

4. void wr3(char * fname)
{
    ofstream file(fname);
    char a[80];
    cout << "input text:" << endl;
    while(1) {
        if(cin.getline(a,80))
            file << a << endl;
        else
            break;
    }
    file.close();
}

5. void wr4(char * fname)
{
    ifstream file(fname, ios::in | ios::nocreate);
    int m,n=0;
    file.seekg(0,ios::end);
    m = file.tellg();
    file.seekg(0);
    char ch;
    file.get(ch);

```

```

        while(ch != EOF) {
            if(ch != ' ' && ch != '\n')
                n++;
            file.get(ch);
        }
        cout << "m=" << m << ", n=" << n << endl;
        file.close();
    }
}

6. void wr5(char * fname, Student a[], int n)
{
    ofstream file(fname, ios::binary);
    for(int i=0; i<n; i++)
        file.write((char*)&a[i], sizeof(Student));
    file.close();
}

7. void wr6(char * fname)
{
    Student x, cmp = {"", "", 0};
    ifstream file(fname, ios::in | ios::nocreate | ios::binary);
    while(1) {
        if(file.read((char*)&x, sizeof(Student))) {
            if(x.grade > cmp.grade) // grade 为学生成绩域
                cmp = x;
        }
        else
            break;
    }
    if(cmp.grade == 0)
        cout << "file is empty!" << endl;
    else
        cout << cmp.num << " " << cmp.name << " " << cmp.grade << endl;
    file.close();
}

```

(五) 按照下面每个题目的要求编写出相应的函数

1. 利用一个字符文件保存 100 以内的所有素数。
2. 利用一个字节文件保存 20 个 100 以内的随机整数,要求保存的所有值各不相同。

```

        while(ch != EOF) {
            if(ch != ' ' && ch != '\n')
                n++;
            file.get(ch);
        }
        cout << "m=" << m << ", n=" << n << endl;
        file.close();
    }
}

6. void wr5(char * fname, Student a[], int n)
{
    ofstream file(fname, ios::binary);
    for(int i=0; i<n; i++)
        file.write((char*)&a[i], sizeof(Student));
    file.close();
}

7. void wr6(char * fname)
{
    Student x, cmp = {"", "", 0};
    ifstream file(fname, ios::in | ios::nocreate | ios::binary);
    while(1) {
        if(file.read((char*)&x, sizeof(Student))) {
            if(x.grade > cmp.grade) // grade 为学生成绩域
                cmp = x;
        }
        else
            break;
    }
    if(cmp.grade == 0)
        cout << "file is empty!" << endl;
    else
        cout << cmp.num << " " << cmp.name << " " << cmp.grade << endl;
    file.close();
}

```

(五) 按照下面每个题目的要求编写出相应的函数

1. 利用一个字符文件保存 100 以内的所有素数。
2. 利用一个字节文件保存 20 个 100 以内的随机整数,要求保存的所有值各不相同。

```

        while(ch != EOF) {
            if(ch != ' ' && ch != '\n')
                n++;
            file.get(ch);
        }
        cout << "m=" << m << ", n=" << n << endl;
        file.close();
    }
}

6. void wr5(char * fname, Student a[], int n)
{
    ofstream file(fname, ios::binary);
    for(int i=0; i<n; i++)
        file.write((char*)&a[i], sizeof(Student));
    file.close();
}

7. void wr6(char * fname)
{
    Student x, cmp = {"", "", 0};
    ifstream file(fname, ios::in | ios::nocreate | ios::binary);
    while(1) {
        if(file.read((char*)&x, sizeof(Student))) {
            if(x.grade > cmp.grade) // grade 为学生成绩域
                cmp = x;
        }
        else
            break;
    }
    if(cmp.grade == 0)
        cout << "file is empty!" << endl;
    else
        cout << cmp.num << " " << cmp.name << " " << cmp.grade << endl;
    file.close();
}

```

(五) 按照下面每个题目的要求编写出相应的函数

1. 利用一个字符文件保存 100 以内的所有素数。
2. 利用一个字节文件保存 20 个 100 以内的随机整数,要求保存的所有值各不相同。

```

        while(ch != EOF) {
            if(ch != ' ' && ch != '\n')
                n++;
            file.get(ch);
        }
        cout << "m=" << m << ", n=" << n << endl;
        file.close();
    }
}

6. void wr5(char * fname, Student a[], int n)
{
    ofstream file(fname, ios::binary);
    for(int i=0; i<n; i++)
        file.write((char*)&a[i], sizeof(Student));
    file.close();
}

7. void wr6(char * fname)
{
    Student x, cmp = {"", "", 0};
    ifstream file(fname, ios::in | ios::nocreate | ios::binary);
    while(1) {
        if(file.read((char*)&x, sizeof(Student))) {
            if(x.grade > cmp.grade) // grade 为学生成绩域
                cmp = x;
        }
        else
            break;
    }
    if(cmp.grade == 0)
        cout << "file is empty!" << endl;
    else
        cout << cmp.num << " " << cmp.name << " " << cmp.grade << endl;
    file.close();
}

```

(五) 按照下面每个题目的要求编写出相应的函数

1. 利用一个字符文件保存 100 以内的所有素数。
2. 利用一个字节文件保存 20 个 100 以内的随机整数,要求保存的所有值各不相同。



```

        while(ch != EOF) {
            if(ch != ' ' && ch != '\n')
                n++;
            file.get(ch);
        }
        cout << "m=" << m << ", n=" << n << endl;
        file.close();
    }
}

6. void wr5(char * fname, Student a[], int n)
{
    ofstream file(fname, ios::binary);
    for(int i=0; i<n; i++)
        file.write((char*)&a[i], sizeof(Student));
    file.close();
}

7. void wr6(char * fname)
{
    Student x, cmp = {"", "", 0};
    ifstream file(fname, ios::in | ios::nocreate | ios::binary);
    while(1) {
        if(file.read((char*)&x, sizeof(Student))) {
            if(x.grade > cmp.grade) // grade 为学生成绩域
                cmp = x;
        }
        else
            break;
    }
    if(cmp.grade == 0)
        cout << "file is empty!" << endl;
    else
        cout << cmp.num << " " << cmp.name << " " << cmp.grade << endl;
    file.close();
}

```

(五) 按照下面每个题目的要求编写出相应的函数

1. 利用一个字符文件保存 100 以内的所有素数。
2. 利用一个字节文件保存 20 个 100 以内的随机整数,要求保存的所有值各不相同。

```

        while(ch!= EOF) {
            if(ch!= ' ' && ch!= '\n')
                n++;
            file.get(ch);
        }
        cout << "m=" << m << ", n=" << n << endl;
        file.close();
    }
}

6. void wr5(char * fname, Student a[], int n)
{
    ofstream file(fname, ios::binary);
    for(int i=0; i<n; i++)
        file.write((char*)&a[i], sizeof(Student));
    file.close();
}

7. void wr6(char * fname)
{
    Student x, cmp = {"", "", 0};
    ifstream file(fname, ios::in | ios::nocreate | ios::binary);
    while(1) {
        if(file.read((char*)&x, sizeof(Student))) {
            if(x.grade > cmp.grade) // grade 为学生成绩域
                cmp = x;
        }
        else
            break;
    }
    if(cmp.grade == 0)
        cout << "file is empty!" << endl;
    else
        cout << cmp.num << " " << cmp.name << " " << cmp.grade << endl;
    file.close();
}

```

(五) 按照下面每个题目的要求编写出相应的函数

1. 利用一个字符文件保存 100 以内的所有素数。
2. 利用一个字节文件保存 20 个 100 以内的随机整数,要求保存的所有值各不相同。

```

        while(ch!= EOF) {
            if(ch!= ' ' && ch!= '\n')
                n++;
            file.get(ch);
        }
        cout << "m=" << m << ", n=" << n << endl;
        file.close();
    }
}

6. void wr5(char * fname, Student a[], int n)
{
    ofstream file(fname, ios::binary);
    for(int i=0; i<n; i++)
        file.write((char*)&a[i], sizeof(Student));
    file.close();
}

7. void wr6(char * fname)
{
    Student x, cmp = {"", "", 0};
    ifstream file(fname, ios::in | ios::nocreate | ios::binary);
    while(1) {
        if(file.read((char*)&x, sizeof(Student))) {
            if(x.grade > cmp.grade) // grade 为学生成绩域
                cmp = x;
        }
        else
            break;
    }
    if(cmp.grade == 0)
        cout << "file is empty!" << endl;
    else
        cout << cmp.num << " " << cmp.name << " " << cmp.grade << endl;
    file.close();
}

```

(五) 按照下面每个题目的要求编写出相应的函数

1. 利用一个字符文件保存 100 以内的所有素数。
2. 利用一个字节文件保存 20 个 100 以内的随机整数,要求保存的所有值各不相同。

```

        while(ch!= EOF) {
            if(ch!= ' ' && ch!= '\n')
                n++;
            file.get(ch);
        }
        cout << "m=" << m << ", n=" << n << endl;
        file.close();
    }
}

6. void wr5(char * fname, Student a[], int n)
{
    ofstream file(fname, ios::binary);
    for(int i=0; i<n; i++)
        file.write((char*)&a[i], sizeof(Student));
    file.close();
}

7. void wr6(char * fname)
{
    Student x, cmp = {"", "", 0};
    ifstream file(fname, ios::in | ios::nocreate | ios::binary);
    while(1) {
        if(file.read((char*)&x, sizeof(Student))) {
            if(x.grade > cmp.grade) // grade 为学生成绩域
                cmp = x;
        }
        else
            break;
    }
    if(cmp.grade == 0)
        cout << "file is empty!" << endl;
    else
        cout << cmp.num << " " << cmp.name << " " << cmp.grade << endl;
    file.close();
}

```

(五) 按照下面每个题目的要求编写出相应的函数

1. 利用一个字符文件保存 100 以内的所有素数。
2. 利用一个字节文件保存 20 个 100 以内的随机整数,要求保存的所有值各不相同。

```

        while(ch != EOF) {
            if(ch != ' ' && ch != '\n')
                n++;
            file.get(ch);
        }
        cout << "m=" << m << ", n=" << n << endl;
        file.close();
    }
}

6. void wr5(char * fname, Student a[], int n)
{
    ofstream file(fname, ios::binary);
    for(int i=0; i<n; i++)
        file.write((char*)&a[i], sizeof(Student));
    file.close();
}

7. void wr6(char * fname)
{
    Student x, cmp = {"", "", 0};
    ifstream file(fname, ios::in | ios::nocreate | ios::binary);
    while(1) {
        if(file.read((char*)&x, sizeof(Student))) {
            if(x.grade > cmp.grade) // grade 为学生成绩域
                cmp = x;
        }
        else
            break;
    }
    if(cmp.grade == 0)
        cout << "file is empty!" << endl;
    else
        cout << cmp.num << " " << cmp.name << " " << cmp.grade << endl;
    file.close();
}

```

(五) 按照下面每个题目的要求编写出相应的函数

1. 利用一个字符文件保存 100 以内的所有素数。
2. 利用一个字节文件保存 20 个 100 以内的随机整数,要求保存的所有值各不相同。